Developer's Guide: QuickTime for Macintosh

Preliminary

Version 2.5

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, ADC will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to ADC.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures, Tables, and Listings

Chapter 1 Movie Toolbox 1-1

Chapter 10 Text Channel Component 10-1

Chapter 11 Movie Data Exchange Components 11-1

About This Book

This book documents version 2.5 of QuickTime for Macintosh. It also describes all the features added or changed in QuickTime for Macintosh since version 1.5, and therefore supersedes all existing documentation for the software releases 1.6.1, 2.0, and 2.1.

The original QuickTime-related *Inside Macintosh* books documented QuickTime for Macintosh through the 1.5 software release. So, your main source for information on programming in QuickTime on the Macintosh is now this book plus the related *Inside Macintosh* books, *Inside Macintosh: QuickTime, Inside Macintosh*: *QuickTime Components*, and *Inside Macintosh: More Macintosh Toolbox*.

A new book describing QuickTime file formats has been added to the QuickTime for Macintosh documentation suite. *QuickTime File Format Specification, May 1996,* supersedes Chapter 4 of *Inside Macintosh: QuickTime*, "Movie Resource Formats." In addition, Appendix B of this book, "QuickTime File Format Changes," provides new information that has not yet been incorporated into the file format specification book.

Many chapters in this book update specific chapters in the *Inside Macintosh* books. Within these update chapters, the section headings correspond to sections found in the *Inside Macintosh* books, wherever possible. However, the update chapters in this book contain only the changed information; you must refer to the *Inside Macintosh* books for all unchanged features. This book also contains chapters that describe completely new areas of functionality that were not documented in the *Inside Macintosh* books.

QuickTime for Windows is documented in a separate set of books. However, you will find in this book some references to differences between QuickTime for the Windows platform and for the Macintosh platform.

Briefly, this book contains the following chapters:

- Chapter 1, "Movie Toolbox," updates Chapter 2 of *Inside Macintosh: QuickTime*. This chapter describes features that are new to the Movie Toolbox or have changed since *Inside Macintosh: QuickTime* was published. These features include the Sprite Toolbox, QT atoms, modifier tracks, and timecode tracks.

- Chapter 2, "Component Manager," updates Chapter 6 of *Inside Macintosh: More Macintosh Toolbox*. This chapter describes four new functions added in QuickTime 2.5.
- Chapter 3, "Image Compression Manager," updates Chapter 3 of *Inside Macintosh: QuickTime*. This chapter describes features that are new to the Image Compression Manager or have changed since *Inside Macintosh: QuickTime* was published. These features include support for ColorSync, alpha channels, timecode data, and asynchronous decompression operations.
- Chapter 4, "Image Compressor Components," updates Chapter 4 of *Inside Macintosh: QuickTime Components*. This chapter describes how compressor and decompressor components have changed in order to support new QuickTime image-compression features.
- Chapter 5, "Image Transcoder Components", describes new image transcoding features introduced with QuickTime 2.5.
- Chapter 6, "Movie Controller Components," updates Chapter 2 of *Inside Macintosh: QuickTime Components*. This chapter describes new movie controller actions, a new function for determining whether a point is inside the control area of a movie, and a new flag returned by the `MCGetControllerInfo` function.
- Chapter 7, "Sequence Grabber Components," updates Chapter 5 of *Inside Macintosh: QuickTime Components*. This chapter provides information about new sequence-grabber features; in particular, the concept of a sequence grabber output.
- Chapter 8, "Sequence Grabber Channel Components," updates Chapter 6 of *Inside Macintosh: QuickTime Components*. This chapter describes how sequence grabber channel components have changed in order to support new sequence-grabber functionality.
- Chapter 9, "Video Digitizer Components," updates Chapter 8 of *Inside Macintosh: QuickTime Components*. This chapter describes new video digitizer component features, including support for timecode tracks.
- Chapter 10, "Text Channel Components," describes new text channel components and text handling features introduced with QuickTime 2.5.
- Chapter 11, "Movie Data Exchange Components," updates Chapter 9 of *Inside Macintosh: QuickTime Components*. This chapter presents information about new data import and export features in QuickTime.

- Chapter 12, "Derived Media Handler Components," updates Chapter 10 of *Inside Macintosh: QuickTime Components*. This chapter describes changes that affect derived media handler components.
- Chapter 13, "Tween Media Handler Components," describes the tween media handler and tween components introduced with QuickTime 2.5.
- Chapter 14, "Sprite Media Handler," describes the sprite media handler introduced with QuickTime 2.5.
- Chapter 15, "Preview Components," updates Chapter 12 of *Inside Macintosh: QuickTime Components*. This chapter describes new support for single-fork movie previews.
- Chapter 16, "Data Handler Components," describes the interface that must be supported by QuickTime data handler components.
- Chapter 17, "Graphics Importer Components," describes new components for opening, displaying, and saving still images.
- Chapter 18, "QuickTime Settings Control Panel" describes new features of the control panel, including CD-ROM AutoStart.
- Chapter 19, "QuickTime Music Architecture," describes the music architecture introduced with QuickTime 2.0, as well as new features in QuickTime 2.5.
- Appendix A, "General MIDI Reference," lists the instrument numbers, drum kit numbers, and MIDI kit names defined by the General MIDI specification.
- Appendix B, "QuickTime File Format Changes," contains changes and additions to the Motion JPEG and YUV file formats as documented in *QuickTime File Format Specification, May 1996*. It also provides information about the QuickTime image file format introduced with QuickTime 2.5.

Format of an Update Chapter

The update chapters in this book follow a standard structure where possible. Each chapter begins with an overview of the new features. Some of the chapters give programming examples, then all the constants, data types, and functions are described in a reference section. For example, Chapter 1, "Movie Toolbox," contains only three main sections, because only three apply:

- "New Features of the Movie Toolbox." This section gives background information about the new or changed features in the Movie Toolbox since the 1.5 software release, which was documented in *Inside Macintosh: QuickTime*,
- "Using the Movie Toolbox." This section describes how to use the new features of the Movie Toolbox.
- "Movie Toolbox Reference." This section describes all the constants, data structures, and functions in the Movie Toolbox. Each function description also follows a standard format, which gives the function declaration and description of every parameter.

Format of a New Chapter

This book also contains chapters that describe completely new areas of functionality that were not documented in the *Inside Macintosh* books. These chapters follow a standard structure. Each begins with an overview, then some chapters give programming examples, then all the constants, data types, and functions are described in a reference section. For example, Chapter 14, "Sprite Media Handler," contains these sections:

- "About the Sprite Media Handler." This section gives you a general overview of the sprite media handler and the features it provides.
- "Using the Sprite Media Handler." This section describes the tasks you can accomplish using the sprite media handler, and gives programming examples.
- "Sprite Media Handler Reference." This section describes the constants, data types, and functions for use with the sprite media handler. Each function description also follows a standard format, which gives the function declaration and description of every parameter.

Conventions Used in This Book

This book provides various conventions to present information. Words that require special treatment appear in specific fonts or font styles. Certain types of

information, such as parameter blocks, use special fonts so that you can scan them quickly.

Special Fonts

All code listings, reserved words, and the names of actual data structures, constants, fields, parameters, and functions are shown in Letter Gothic (`this is Letter Gothic`).

Words that appear in **boldface** are key terms or concepts that are defined in the glossary.

Types of Notes

There are several types of notes used in this book.

Note
A note like this contains information that is interesting but not essential to an understanding of the main text. ◆

IMPORTANT
A note like this contains information that is essential for an understanding of the main text. ▲

▲ **WARNING**
A warning like this indicates potential problems that you should be aware of as you design your game. Failure to heed these warnings could result in system crashes or loss of data. ▲

Development Environment

The functions described in this book are available using C interfaces. How you access them depends on the development environment you are using.

Code listings in this book are shown in ANSI C. They suggest methods of using various functions and illustrate techniques for accomplishing particular tasks.

Although most code listings have been compiled and tested, Apple Computer Inc., does not intend for you to use these code samples in your application.

For More Information

The *Apple Developer Catalog* (ADC) is Apple Computer's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple computer platforms. Customers receive the *Apple Developer Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. ADC offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *Apple Developer Catalog*, contact

Apple Developer Catalog
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	ORDER.ADC
Internet	order.adc@applelink.apple.com

Movie Toolbox

Contents

This chapter discusses the changes to the Movie Toolbox as documented in Chapter 2 of *Inside Macintosh: QuickTime*.

New Features of the Movie Toolbox

Preloading Tracks

There are occasions when it may be useful for you to preload some or all of a track's media data into memory. For example, if you are developing an application that plays several movies at once, you may want to load the smaller movies into memory in order to reduce CD-ROM seek activity. Text tracks, which are typically rather small, are good candidates for preloading; in many cases you can load a movie's entire text track into memory. Another good use of preloading is to preload small music tracks that play over scene changes, giving the interactive experience a more continuous feel.

QuickTime 2.0 expanded your options for preloading tracks. In the past, applications could use the `Load...IntoRAM` functions to preload a movie, track, or media. Now, you can establish preloading guidelines as part of a track's definition. The Movie Toolbox then automatically preloads the track, according to those guidelines, every time the movie is played, and without any special effort by applications. You establish these preloading guidelines by calling the new `SetTrackLoadSettings` function, see "Enhancing Movie Playback Performance" (page 1-65) for more information about this function). Note that the preloading information is preserved in flattened movies.

Hints

QuickTime 1.6.1 and 2.0 defined several new movie and media playback hints. These new hints work with the `SetMoviePlayHints` and `SetMediaPlayHints` functions.

`hintsDontPurge`

Instructs the Movie Toolbox not to dispose of movie data after playing it. The Movie Toolbox leaves the data in memory, in a purgeable handle. This can enhance the playback of small

movies that are looping. However, it may consume large amounts of memory and therefore affect the performance of the Memory Manager. Use this hint carefully.

`hintsInactive`

Tells the Movie Toolbox that the movie is not in an active window. This can allow the Movie Toolbox to more efficiently allocate scarce system resources. The movie controller component automatically sets this hint for all movies it manages.

`hintsHighQuality`

Specifies that the given movie or media should render the highest quality. For example, the video media handler turns off fast dithering and uses high-quality dithering.

Because rendering at the highest quality takes much more time and memory than rendering at a lesser quality, this mode is usually not appropriate for real-time playback. However, since this mode generates higher quality images, it is useful when recompressing.

Data References

The Movie Toolbox now fully supports a media that refers to data in more than one file. In the past, a media was restricted to a single data file. By allowing a single media to refer to more than one file, the Movie Toolbox allows better playback performance and easier editing, primarily by reducing the number of tracks in a movie. Use the new `SetMediaDefaultDataRefIndex` function (page 1-83) to control which of a media's files you access when you add new samples.

Timecode Media Handler

QuickTime 2.0 introduced support for timecode tracks. Timecode tracks allow you to store external timecode information, such as SMPTE timecode, in your QuickTime movies. QuickTime now provides a new timecode media handler that interprets the data in these tracks.

See "Timecode Media Handler Functions" (page 1-99) for information about these functions.

Track References

While QuickTime has always allowed the creation of movies that contain more than one track, it has not been able to specify relationships between those tracks. **Track references** are a new feature of QuickTime that allow you to relate a movie's tracks to one another. The QuickTime track-reference mechanism supports many-to-many relationships. That is, any movie track may contain one or more track references, and any track may be related to one or more other tracks in the movie.

Track references can be useful in a variety of ways. For example, track references can be used to relate timecode tracks to other movie tracks. (See "Timecode Media Handler" for more information about timecode tracks.) You might consider using track references to identify relationships between video and sound tracks, identifying the track that contains dialog and the track that contains background sounds, for example. Another use of track references is to associate one or more text tracks that contain subtitles with the appropriate audio track or tracks.

Every movie track contains a list of its track references. Each track reference identifies another, related track. That related track is identified by its track identifier. The track reference itself contains information that allows you to classify the references by type. This type information is stored in an `OSType` data type. You are free to specify any type value you want—note, however, that Apple has reserved all lower-case type values.

You may create as many track references as you want, and you may create more than one reference of a given type. Each track reference of a given type is assigned an index value. These index values start at 1 for each different reference type. The Movie Toolbox maintains these index values so that they always start at 1 and count by 1.

See "Working With Track References" (page 1-76) for detailed descriptions of the Movie Toolbox functions that allow you to work with track references.

Modifier Tracks

The addition of **modifier tracks** in QuickTime 2.1 introduced new capabilities for creating dynamic movies. For example, instead of playing video in a normal way, a video track can send its image data to a sprite track. The sprite track then uses that video data to replace the image of one of its sprites. When the movie is played, the video track appears as a sprite.

Modifier tracks are not a new type of track. Instead, they are a new way of using the data in existing tracks. A modifier track does not present its data, but instead sends it to another track which uses the data to modify how it presents its own data. Any track can be either a sender or a presenter, but not both. Previously, all tracks were presenters.

Another use of modifier tracks is to store a series of sound volume levels. These sound levels can be sent to a sound track as it plays to dynamically adjust the volume. A similar use of modifier tracks is to store location and size information. This data can be sent to a video track to cause it to move and resize as it plays.

Because a modifier track can send its data to more than one track, you can easily synchronize actions between multiple tracks. For example, a single modifier track containing matrices as its samples can make two separate video tracks follow the same path.

See "Creating Movies With Modifier Tracks" (page 1-21) for more information about using modifier tracks. See "Tween Media" (page x-x) for more information.

Limitations of Spatial Modifier Tracks

A modifier track can cause a track to move outside of its original bounds. This could cause problems, as applications don't expect a QuickTime movie's dimensions or location to change over time. To ensure that the movie maintains a constant location and size, the Movie Toolbox limits the area a spatially modified track can be displayed in. A movie's "natural" shape is defined by the region returned by `GetMovieBoundsRgn`. The Movie Toolbox clips all spatially modified tracks against the region returned by `GetMovieBoundsRgn`. This means that a track can move outside of its initial bounds, but it cannot move beyond the combined initial bounds of all tracks in the movie. Areas uncovered by a moving track are handled by the Movie Toolbox in the same way as areas uncovered by tracks with empty edits. For more information about how QuickTime handles uncovered areas, see the description of the `SetMovieCoverProcs` function on page 2-156 of *Inside Macintosh: QuickTime.*

If it is necessary for a to track move through a larger area than that defined by the movie's bounds region, the movie's bounds region can be enlarged to any desired size by creating a spatial track (such as a video track) of the desired size but with no data. As long as the track is enabled, it will contribute to the bounds of the movie.

Media Handler Support

The following media handlers support sending their data to other tracks: Video, Base, and Tween.

The Sound, Music and 3D media handlers do not support sending their data to other tracks.

Not all media handlers support all input types. Media handlers can decide which input types to support. Table 1-1 lists the input types supported by each Apple-supplied media handler:

Table 1-1 Input Types Supported by Each Apple-supplied Media Handler

	Video	Text	Sound	MPEG	Music	Sprite	Time Code	3D
Matrix	✓	✓		✓		✓	✓	✓
Graphics Mode	✓	✓		✓		✓	✓	✓
Clip	✓	✓		✓		✓	✓	✓
Volume			✓	✓	✓			
Balance			✓	✓	✓			
Sprite Image						✓		✓
3D Sound			✓		✓			

Data Handler Components

QuickTime 2.0 includes a new, memory-based data handler. This data handler component works with movie data that is stored in memory, in a handle, rather than in a file. This data handler has a component subtype value of `HandleDataHandlerSubType` (`'hndl'`).

To create a movie that uses the handle data handler, set the data reference type to `HandleDataHandlerSubType` when you call the `NewTrackMedia` function. Note that the movie data in memory is not automatically saved with the movie. If you want to save the data that is in memory, use the `FlattenMovie` or `InsertTrackSegment` functions to copy the data from memory to a file.

The handle data handler does not use aliases as its data reference, and therefore does not use alias handles. Rather, it uses 4-byte memory handles as its data reference. The data reference contains the actual handle that stores the needed data. If you pass a handle value of `nil`, the data handler allocates and manages the handle for you. If you pass a non-`nil` handle value, the data handler uses your handle. It is then your responsibility to manage the handle, and dispose of it when appropriate. Note that a single handle may be shared by several data handler components. Whenever new data is added, the data handler resizes the handle to accommodate new data.

Although data handler components have existed since QuickTime 1.0, their interface is publicly defined for the first time in QuickTime 2.0. If you are interested in developing a data handler, refer to the chapter "Data Handler Components" later in this document.

Sprite Toolbox

The Sprite Toolbox is a set of data types and functions you can use to add sprite-based animation to an application. The Sprite Toolbox handles invalidating appropriate areas as sprite properties change, the composition of sprites and their background on an offscreen buffer, and the transfer of the result to the screen or an alternate destination.

To create a sprite track in a QuickTime movie, you use the sprite media handler, which, in turn, makes use of the Sprite Toolbox. For information on how to use the sprite media handler, see the chapter "Sprite Media Handler" in this book.

This section describes the characteristics that govern creating sprite animation in an application.

Sprite Characteristics

Sprite animation differs substantially from traditional video animation. With traditional video animation, you describe a frame by specifying the color of each pixel. By contrast, with **sprite animation**, you describe a frame by specifying which **sprites**, or images, taken from a finite set of images, appear at various locations.

You can think of a sprite animation as a play. In a QuickTime movie, the sprite track bounds are the stage; in an application, a sprite world is the stage. The

background is the play's set; the background may be a single solid color, an image, or a combination of images. The sprites are the actors in the play.

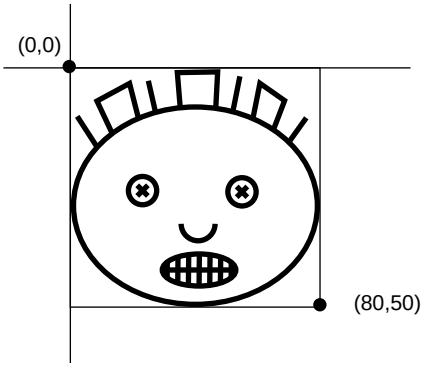
A sprite has properties that describe its location and appearance at a given point in time. During the course of an animation, you modify a sprite's properties to cause it to change its appearance and move around the set. For sprites in a sprite world, you modify a sprite's properties by calling the `SetSpriteProperty` function, passing a constant to indicate which property you want to modify. `SetSpriteProperty` invalidates the appropriate portions of the sprite world, which are redrawn when `SpriteWorldIdle` is called. For sprites in a sprite track, you modify a sprite property by creating an override sample of the appropriate type.

Each sprite has a corresponding image. During the animation, you can change a sprite's image. For example, you can assign a series of images to a sprite in succession to perform cell-based animation. For sprites in a sprite world, you control a sprite's image by setting the sprite's `kSpritePropertyImageDescription` and `kSpritePropertyImageDataPtr` properties.

For sprites in a sprite track, all sprite images are stored in the sprite track's key frame sample. This allows the sprites in the sprite track to share images. A sprite's image index (`kSpritePropertyImageIndex`) specifies the sprite's current image in the pool of available images. All images assigned to a sprite must share the same image description.

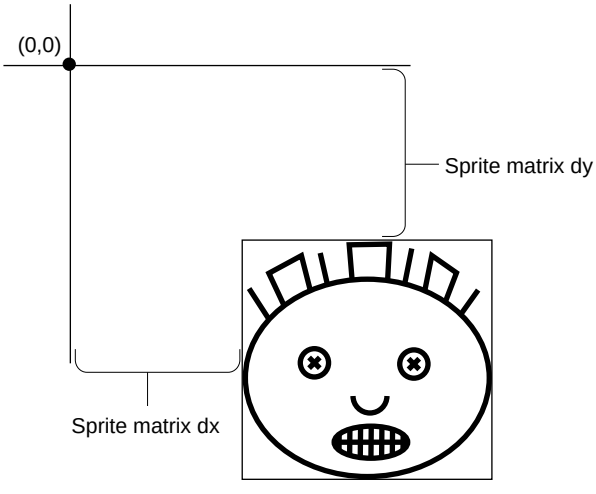
A sprite's matrix property (`kSpritePropertyMatrix`) describes the sprite's location and scaling within its sprite world or sprite track. A sprite's local coordinate system is defined by its natural bounding box, as shown in Figure 1-1.

Figure 1-1 Local coordinate system of a sprite



A sprite's display coordinate system — where a sprite is displayed within a sprite world or a sprite track — is defined by the sprite's matrix, as shown in Figure 1-2.

Figure 1-2 Display coordinate system of a sprite



By modifying a sprite's matrix, you can modify the sprite's location so that it appears to move in a smooth path on the screen or so that it jumps from one place to another. You can modify a sprite's size so that it shrinks, grows, or stretchs. Depending on which image compressor is used to create the sprite images, other transformations, such as rotation, may be supported as well. Translation only matrices provide the best performance.

A sprite's layer property (`kSpritePropertyLayer`) is a numeric value that specifies a sprite's layer in the animation. Sprites with lower layer numbers appear in front of sprites with higher layer numbers. To designate a sprite as a background sprite, you should assign it the special layer number `kBackgroundSpriteLayerNum`.

A sprite's visible property (`kSpritePropertyVisible`) specifies whether or not the sprite is visible. To make a sprite visible, you set the sprite's visible property to `true`.

A sprite's graphics mode property (`kSpritePropertyGraphicsMode`) specifies a graphics mode and blend color that indicates how to blend a sprite with any sprites behind it and with the background. To set a sprite's graphics mode, you call `SetSpriteProperty`, passing a pointer to a `ModifierTrackGraphicsModeRecord` structure.

Three other properties, `kSpriteTrackPropertyBackgroundColor`, `kSpriteTrackPropertyOffscreenBitDepth`, and `kSpriteTrackPropertySampleFormat`, describe properties of a sprite track in a QuickTime movie. These properties are discussed in more detail in Chapter 14, "Sprite Media Handler."

Sprite World Characteristics

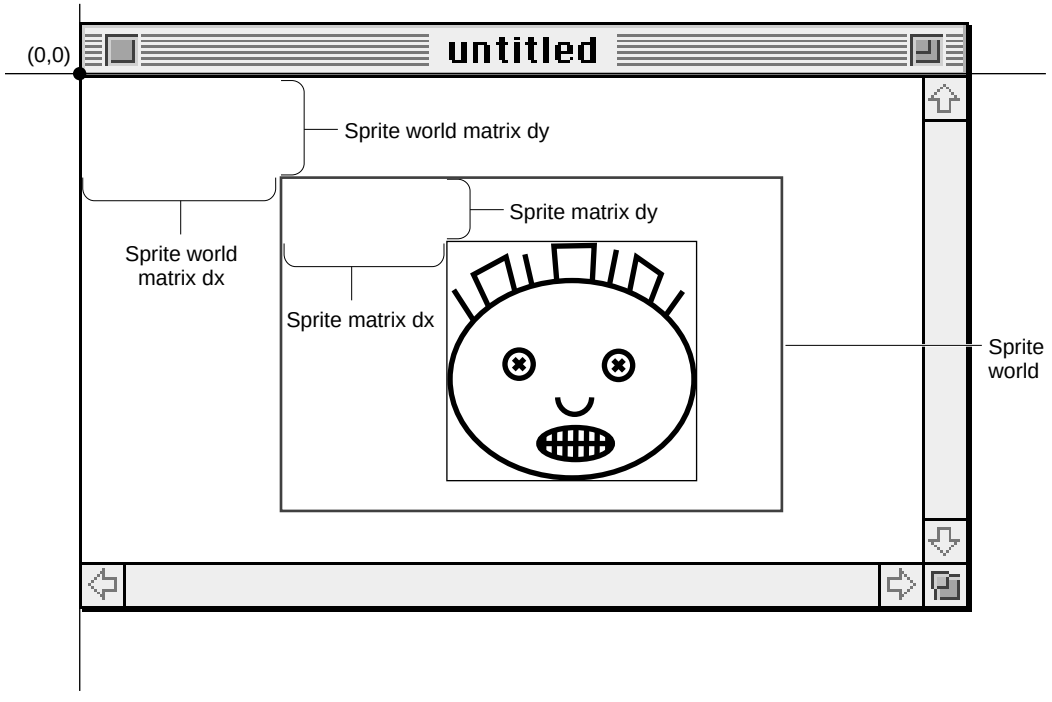
A **sprite world** is a graphics world for a sprite animation. To create a sprite animation in an application, you must first create a sprite world. You do not need to create a sprite world to create a sprite track in a QuickTime movie.

Once you have created a sprite world, you create sprites associated with that sprite world. You can think of a sprite world as a stage on which your sprites perform. When you dispose of a sprite world, its associated sprites are disposed of as well.

When you call `SetSpriteProperty` to modify a property of a sprite, `SetSpriteProperty` invalidates the appropriate regions of the sprite world. When your application calls `SpriteWorldIdle`, the sprite world redraws its invalid regions. A sprite's sprite world coordinate system is defined by

translating the sprite's display coordinate system by the sprite world's matrix, as shown in Figure 1-3.

Figure 1-3 Sprite world coordinate system



To achieve the best performance for your sprite animation, you should observe the following guidelines when creating a sprite world.

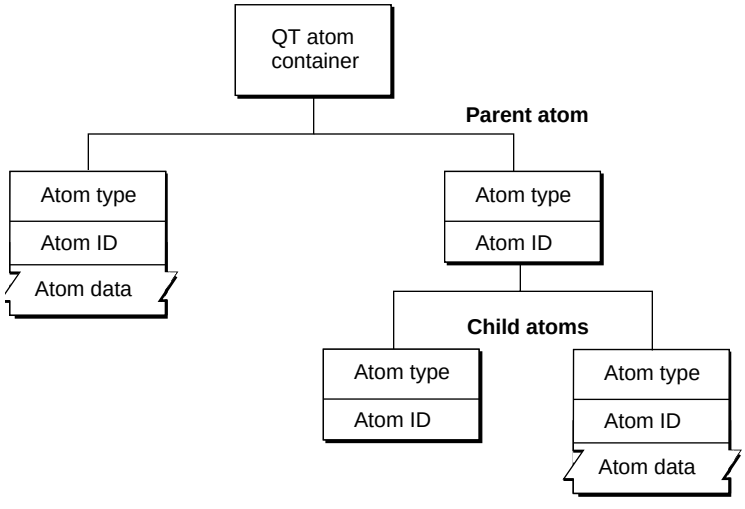
- When you create a graphics world to be used for your sprite world, you will achieve the best performance if the graphics world's dimensions are a multiple of 16 pixels.
- Your sprite layer graphics world and background graphics world should both be the same size and depth as the destination of your sprite animation.
- Use translation-only matrices for creating sprite worlds and sprites.
- Do not set a clipping region for your sprite world.

- Call the `SpriteWorldIdle` function frequently.
- Avoid clipping sprites with the sprite world boundry.
- Use the Animation compressor to create sprites with transparent areas.

QT Atoms

A **QT atom container** is a basic structure for storing information in QuickTime. You can use a QT atom container to construct arbitrarily complex hierarchical data structures. You can think of a newly created QT atom container as the root of a tree that contains no children. A QT atom container contains **QT atoms** (Figure 1-4). Each QT atom contains either data or other atoms. If a QT atom contains other atoms, it is a **parent atom** and the atoms it contains are its **child atoms**. If a QT atom contains data, it is called a **leaf atom**.

Figure 1-4 QT atom container with parent and child atoms



Each QT atom has an offset that describes the atom's position within the QT atom container. In addition, each QT atom has a type and an ID. The atom type describes the kind of information the atom represents. The atom ID is used to differentiate child atoms of the same type with the same parent; an atom's ID

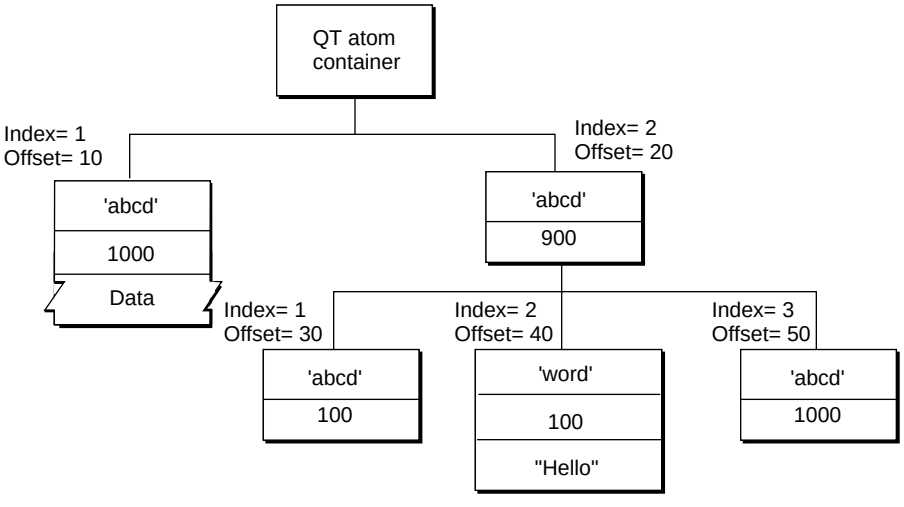
must be unique for a given parent and type. In addition to the atom ID, each atom has a one-based index that describes its order relative to other child atoms of the same parent. You can uniquely identify a QT atom in three ways:

- by its offset
- by its parent atom, type, and index
- by its parent atom, type, and ID

You can store and retrieve atoms in a QT atom container by index, ID, or both. For example, to use a QT atom container as a dynamic array or tree structure, you can store and retrieve atoms by index. To use a QT atom container as a database, you can store and retrieve atoms by ID. Or, you can create store and retrieve atoms using ID and index to create an arbitrarily complex, extensible data structure.

Figure 1-5 shows a QT atom container that has two child atoms. The first child atom (offset = 10) is a leaf atom that has an atom type of `'abcd'`, an ID of 1000, and an index of 1. The second child atom (offset = 20) has an atom type of `'abcd'`, an ID of 900, and an index of 2. Because the two child atoms have the same type, they must have different IDs. The second child atom is also a parent atom of three atoms.

Figure 1-5 QT atom container example



The first child atom (offset = 30) has an atom type of `'abcd'`, an ID of 100, and an index of 1. It does not have any children, nor does it have data. The second child atom (offset = 40) has an atom type of `'word'`, an ID of 100, and an index of 2. The atom has data, so it is a leaf atom. The second atom (offset = 40) has the same ID as the first atom (offset = 30), but a different atom type. The third child atom (offset = 50) has an atom type of `'abcd'`, an ID of 1000, and an index of 3. Its atom type and ID are the same as that of another atom (offset = 10) with a different parent.

For more information about the structure of QT atoms and atom containers, see the book *QuickTime File Format Specification, May 1996*.

As a developer, you do not need to parse QT atoms yourself. Instead, you can use the QT atom functions to create atom containers, add atoms to and remove atoms from atom containers, search for atoms in atom containers, and retrieve data from atoms in atom containers.

Most QT atom functions take two parameters to specify a particular atom: the atom container that contains the atom and the offset of the atom in the atom container data structure. You obtain an atom's offset by calling either `QTFindChildByID` or `QTFindChildByIndex`. An atom's offset may be invalidated if the QT atom container that contains it is modified.

When calling any QT atom function for which you specify a parent atom as a parameter, you can pass the constant `kParentAtomIsContainer` as an atom offset to indicate that the specified parent atom is the atom container itself. For example, you call the `QTFindChildByIndex` function and pass `kParentAtomIsContainer` constant for the parent atom parameter to indicate that the requested child atom is a child of the atom container itself.

Using the Movie Toolbox

This section describes new or changed operations your application may perform with the Movie Toolbox. The following topics are discussed:

- "Loading a Movie" describes how to load a movie from a non-HFS volume.
- "Creating a Movie With Modifier Tracks" describes how media handlers can send their data to another media handler rather than presenting their media directly.

- "Creating and Initializing a Sprite World" describes the steps for creating and initializing a sprite world.
- "Creating and Initializing Sprites" discusses how to create sprites and add them to a sprite world.
- "Animating Sprites" shows how to animate sprites by altering their properties.
- "Disposing of a Sprite Animation" shows how to dispose of sprite-related data structures when you have finished displaying a sprite animation.
- "Sprite Hit Testing" discusses how to perform hit testing operations for sprites.
- "Creating and Disposing of Atom Containers" discusses how to create a new QT atom container and how to dispose of a QT atom container when you have finished using it.
- "Creating New Atoms" describes how to create new atoms and insert them into an atom container.
- "Copying Existing Atoms" discusses how to copy and manipulate existing atoms within an atom container.
- "Retrieving Atoms From an Atom Container" shows how to retrieve atoms and their data from an atom container.
- "Modifying Atoms" shows how to modify the data of atoms stored in an atom container.
- "Removing Atoms From an Atom Container" describes how to remove atoms from an atom container.

Loading a Movie

QuickTime 2.0 made the data handler component interface available to developers. Data handlers provide a way to access data stored in any location, in any kind of container. Using a data handler, you can access data on a Macintosh hard disk, stored in memory, or stored on a network volume on a non-HFS volume. Although data handlers allow a movie data to be stored on any kind of device, before QuickTime 2.0, the movie resource had to be stored on an HFS volume. QuickTime 2.1 provides you with a new function, named `NewMovieFromDataRef`, that allows a movie to be created from any device that

has a corresponding data handler. Use the `NewMovieFromDataRef` function (page 1-59) when you need to instantiate movies from other types of devices.

Creating Movies With Modifier Tracks

QuickTime 2.1 provides additional functionality for media handlers. By way of modifier tracks, media handlers can now send their data to another media handler rather than presenting their media directly. See "Modifier Tracks" (page 1-9) earlier in this chapter for a complete discussion of this feature.

To create a movie with modifier tracks, first you create a movie with all the desired tracks, then you create the modifier track. To link the modifier tracks to the track that it will modify, use the existing `AddTrackReference` function as shown in the following code:

```
long addedIndex;

AddTrackReference(aVideoTrack, aModifierTrack, kTrackModifierReference,
      &addedIndex);
```

The reference doesn't completely describe the modifier track's relationship to the track it modifies. Instead, the reference simply tells the modifier track to send its data to the specified track. The receiving track doesn't know what it should do with that data. A single track may also be receiving data from more than one modifier track. To describe how each modifier input should be used, each track's media also has an **input map**. The media's input map describes how the data being sent to each input of a track should be interpreted by the receiving track. After creating the reference, it is necessary to update the receiving track's media input map. When `AddTrackReference` is called, it returns the index of the reference added. That index is the index of the input which needs to be described in the media input map. If the modifier track created above contained regions to change the shape of the video track, the following code updates the input map appropriately:

```
QTAtomContainer inputMap;
QTAtom inputAtom;
OSType inputType;
Media aVideoMedia = GetTrackMedia(aVideoTrack);
GetMediaInputMap (aVideoMedia, &inputMap);
QTInsertChild(inputMap, kParentAtomIsContainer, kTrackModifierInput,
        addedIndex, 0,0, nil, &inputAtom);
```

```
inputType = kTrackModifierTypeClip;
QTInsertChild (inputMap, inputAtom, kTrackModifierType, 1, 0,
        sizeof(inputType), &inputType, nil);
SetMediaInputMap(aVideoMedia, inputMap);
QTDisposeAtomContainer(inputMap);
```

The media input map allows you to store additional information for each input. In the preceding example, only the type of the input is specified. In other types of references, you may need to specify additional data.

When a modifier track is playing an empty track edit, is disabled or deleted, all receiving tracks are notified that the track input is inactive. When an input becomes inactive, it resets to its default value. For example, if a track was receiving data from a clip modifier track and that input becomes inactive, the shape of the track will revert to the shape it would have if there were no clip modifier track.

Creating and Initializing a Sprite World

To create a sprite animation in an application, first create a sprite world to contain your sprites. To do this, perform the following steps:

- Allocate a sprite layer graphics world that corresponds to the size and bit depth of your destination graphics world.
- If you plan to have a background image behind your sprites that is static or that changes infrequently, create a background graphics world that is the same size and depth as the sprite layer graphics world. You do not need to do this if you plan to have a solid background color behind your sprites. Animations that use a solid background color require less memory and perform slightly better than animations that use a background image.
- Call `LockPixels` on the pixel maps of the sprite layer and background graphics worlds. These graphics worlds must remain valid for the lifetime of the sprite world.
- Call the `NewSpriteWorld` function to create the new sprite world.

The sample code function `CreateSpriteStuff`, Listing 1-1, calculates the bounds of the destination window and calls `NewGWorld` to create a new sprite layer graphics world. Then, it calls `LockPixels` to lock the pixel map of the sprite layer graphics world.

Next, `CreateSpriteStuff` calls `NewSpriteWorld` to create a new sprite world, passing the destination graphics world (`WindowPtr`) and the sprite layer graphics world. `CreateSpriteStuff` passes a background color to `NewSpriteWorld` instead of specifying a background graphics world. The newly created sprite world is returned in the global variable `gSpriteWorld`.

Finally, `CreateSpriteStuff` calls the sample code function `CreateSprites` to create sprites to populate the sprite world with sprites.

Listing 1-1 Creating a sprite world

```
// global variables
GWorldPtr spritePlane = nil;
SpriteWorld gSpriteWorld = nil;
Rect gBounceBox;
RGBColor gBackgroundColor;

void CreateSpriteStuff (Rect *windowBounds, CGrafPtr windowPtr)
{
    OSErr err;
    Rect bounds;

    // calculate the size of the destination
    bounds = *windowBounds;
    OffsetRect (&bounds, -bounds.left, -bounds.top);
    gBounceBox = bounds;
    InsetRect (&gBounceBox, 16, 16);

    // create a sprite layer graphics world with a bit depth of 16
    NewGWorld (&spritePlane, 16, &bounds, nil, nil, useTempMem);
    if (spritePlane == nil)
        NewGWorld (&spritePlane, 16, &bounds, nil, nil, 0);

    if (spritePlane)
    {
        LockPixels (spritePlane->portPixMap);
        gBackgroundColor.red = gBackgroundColor.green =
            gBackgroundColor.blue = 0;

        // create a sprite world
```

```
        err = NewSpriteWorld (&gSpriteWorld, (CGrafPtr)windowPtr,
            spritePlane, &gBackgroundColor, nil);

        // create sprites
        CreateSprites ();
    }
}
```

Creating and Initializing Sprites

Once you have created a sprite world, you can create sprites within it. To do this, you must first obtain image descriptions and image data for your sprites. This image data may be any image data that has been compressed using QuickTime's Image Compression Manager.

You create sprites and add them to your sprite world using the `NewSprite` function. If you want to create a sprite that is drawn to the background graphics world, you should specify the constant `kBackgroundSpriteLayerNum` for the `layer` parameter.

The sample code function `CreateSprites`, shown in Listing 1-2, creates the sprites for the sample application. First, the function initializes some global arrays with position and image information for the sprites.

Next, `CreateSprites` iterates through all the sprite images, preparing each image for display. For each image, `CreateSprites` calls the sample code function `MakePictTransparent` function, which strips any surrounding background color from the image. `MakePictTransparent` does this by using the animation compressor to recompress the PICT images using a key color. Then, `CreateSprites` calls `ExtractCompressData`, which extracts the compressed data from the PICT image. This is one technique for creating compressed images; there are other, more optimized ways to store and retrieve sprite images.

Once the images have been prepared, `CreateSprites` calls `NewSprite` to create each sprite in the sprite world. `CreateSprites` creates each sprite in a different layer.

Listing 1-2 Creating sprites

```
// constants
#define kNumSprites 4
#define kNumSpaceShipImages 24
#define kBackgroundPictID 158
#define kFirstSpaceShipPictID (kBackgroundPictID + 1)
#define kSpaceShipWidth 106
#define kSpaceShipHeight 80

// global variables
SpriteWorld gSpriteWorld = nil;
Sprite gSprites[kNumSprites];
Rect gDestRects[kNumSprites];
Point gDeltas[kNumSprites];
short gCurrentImages[kNumSprites];
Handle gCompressedPictures[kNumSpaceShipImages];
ImageDescriptionHandle gImageDescriptions[kNumSpaceShipImages];

void CreateSprites (void)
{
    long i;
    Handle compressedData = nil;
    PicHandle picture;
    CGrafPtr savePort;
    GDHandle saveGD;
    OSErr err;
    RGBColor keyColor;

    SetRect (&gDestRects[0], 132, 132, 132 + kSpaceShipWidth,
        132 + kSpaceShipHeight);
    SetRect (&gDestRects[1], 50, 50, 50 + kSpaceShipWidth,
        50 + kSpaceShipHeight);
    SetRect (&gDestRects[2], 100, 100, 100 + kSpaceShipWidth,
        100 + kSpaceShipHeight);
    SetRect (&gDestRects[3], 130, 130, 130 + kSpaceShipWidth,
        130 + kSpaceShipHeight);

    gDeltas[0].h = -3;
    gDeltas[0].v = 0;
    gDeltas[1].h = -5;
```

```
    gDeltas[1].v = 3;
    gDeltas[2].h = 4;
    gDeltas[2].v = -6;
    gDeltas[3].h = 6;
    gDeltas[3].v = 4;

    gCurrentImages[0] = 0;
    gCurrentImages[1] = kNumSpaceShipImages / 4;
    gCurrentImages[2] = kNumSpaceShipImages / 2;
    gCurrentImages[3] = kNumSpaceShipImages * 4 / 3;

    keyColor.red = keyColor.green = keyColor.blue = 0xFFFF;

    // recompress PICT images to make them transparent
    for (i = 0; i < kNumSpaceShipImages; i++)
    {
        picture = (PicHandle) GetPicture (i + kFirstSpaceShipPictID);
        DetachResource ((Handle)picture);

        MakePictTransparent (picture, &keyColor);
        ExtractCompressData (picture, &gCompressedPictures[i],
            &gImageDescriptions[i]);
        HLock (gCompressedPictures[i]);

        KillPicture (picture);
    }

    // create the sprites for the sprite world
    for (i = 0; i < kNumSprites; i++)
    {
        MatrixRecord matrix;

        SetIdentityMatrix (&matrix);

        matrix.matrix[2][0] = ((long)gDestRects[i].left << 16);
        matrix.matrix[2][1] = ((long)gDestRects[i].top << 16);

        err = NewSprite (&(gSprites[i]), gSpriteWorld,
            gImageDescriptions[i],* gCompressedPictures[i],
```

```
            &matrix, true, i);
    }
}
```

Animating Sprites

To animate a sprite, you use the `SetSpriteProperty` function to change one or more of the sprite's properties, such as its matrix, layer, or image data. In addition to modifying a property, `SetSpriteProperty` invalidates the appropriate areas of the sprite's sprite world.

The `SpriteWorldIdle` function is responsible for redrawing a sprite world's invalid regions. Your application should call this function after modifying sprite properties to give the sprite world the opportunity to redraw.

Listing 1-3 shows the sample application's `main` function. It performs all of the application's initialization tasks, including initializing the sprite world and its sprites. It displays the window and loops until the user clicks the button in the window. To perform the animation, each time through the loop, `main` calls the sample code function `MoveSprites` to modify the properties of the sprites and then calls `SpriteWorldIdle` to give the sprite world the opportunity to redraw its invalid areas.

Listing 1-3 The `main` function

```
// global variables
SpriteWorld gSpriteWorld = nil;

void main (void)
{

    // ...
    // initialize everything and create a window
    // create a sprite world and the sprites in it
    // show the window
    // ...

    while (!Button())
    {
        // animate the sprites
```

```
        MoveSprites ();
        SpriteWorldIdle (gSpriteWorld, 0, 0);
    }

    // ...
    // dispose of the sprite world and its sprites
    // shut down everything else
    // ...
}
```

The `MoveSprites` function, shown in Listing 1-4, is responsible for modifying the properties of the sprites. For each sprite, the function calls `SetSpriteProperty` twice, once to change the sprite's matrix and once to change the sprite's image data pointer.

Listing 1-4 Animating sprites

```
// constants
#define kNumSprites 4
#define kNumSpaceShipImages 24

// global variables
Rect gBounceBox;
Sprite gSprites[kNumSprites];
Rect gDestRects[kNumSprites];
Point gDeltas[kNumSprites];
short gCurrentImages[kNumSprites];
Handle gCompressedPictures[kNumSpaceShipImages];

void MoveSprites (void)
{
    short i;
    MatrixRecord matrix;

    SetIdentityMatrix (&matrix);

    // for each sprite
    for (i = 0; i < kNumSprites; i++)
    {
        // modify the sprite's matrix
```

```
        OffsetRect (&gDestRects[i], gDeltas[i].h, gDeltas[i].v);

        if ( (gDestRects[i].right >= gBounceBox.right) ||
            (gDestRects[i].left <= gBounceBox.left) )
            gDeltas[i].h = -gDeltas[i].h;

        if ( (gDestRects[i].bottom >= gBounceBox.bottom) ||
            (gDestRects[i].top <= gBounceBox.top) )
            gDeltas[i].v = -gDeltas[i].v;

        matrix.matrix[2][0] = ((long)gDestRects[i].left << 16);
        matrix.matrix[2][1] = ((long)gDestRects[i].top << 16);

        SetSpriteProperty (gSprites[i], kSpritePropertyMatrix, &matrix);

        // change the sprite's image
        gCurrentImages[i]++;
        if (gCurrentImages[i] >= (kNumSpaceShipImages * (i+1)))
            gCurrentImages[i] = 0;
        SetSpriteProperty (gSprites[i], kSpritePropertyImageDataPtr,
            *gCompressedPictures[gCurrentImages[i] / (i+1)] );
    }
}
```

Disposing of a Sprite Animation

When an application has finished displaying a sprite animation, it should do the following things:

- dispose of the sprite world associated with the animation. Disposing of a sprite world automatically destroys the sprites in the sprite world.
- dispose of the sprite image data
- dispose of graphics worlds associated with the sprite animation

In the sample application, `main` calls the sample code function `DisposeEverything` to dispose of sprite-related structures. This function, shown in Listing 1-5, iterates through the sprites, disposing of each sprite's image data. Then, `DisposeEverything` calls `DisposeSpriteWorld` to dispose of the sprite world and all of the sprites in it. Finally, the function calls `DisposeGWorld` to dispose of the graphics world associated with the sprite world.

Listing 1-5 Disposing of sprites and the sprite world

```
// constants
#define kNumSprites 4
#define kNumSpaceShipImages 24

// global variables
SpriteWorld gSpriteWorld = nil;
Sprite gSprites[kNumSprites];
Handle gCompressedPictures[kNumSpaceShipImages];
ImageDescriptionHandle gImageDescriptions[kNumSpaceShipImages];

void DisposeEverything (void)
{
    short i;
    // dispose of the sprite world and associated graphics world
    if (gSpriteWorld)
        DisposeSpriteWorld (gSpriteWorld);

    // dispose of each sprite's image data
    for (i = 0; i < kNumSprites; i++)
    {
        if (gCompressedPictures[i])
            DisposeHandle (gCompressedPictures[i]);
        if (gImageDescriptions[i])
            DisposeHandle ((Handle)gImageDescriptions[i]);
    }
    DisposeGWorld (spritePlane);
}
```

Sprite Hit Testing

The Sprite Toolbox provides two functions for performing hit testing operations with sprites, `SpriteWorldHitTest` and `SpriteHitTest`.

The `SpriteWorldHitTest` function determines whether any sprites exist at a specified location in a sprite world's display coordinate system. This function retrieves the frontmost sprite at the specified location.

The `SpriteHitTest` function determines whether a particular sprite exists at a specified location in the sprite's display coordinate system. This function is

useful for hit testing a subset of the sprites in a sprite world and for detecting multiple sprites at a single location.

You can pass flags to either hit testing function to control the operation more precisely. For example, you may want the hit test operation to detect a sprite whose bounding box contains the specified location. The allowable flags are described in "Flags for Sprite Hit Testing" (page 1-49).

Creating and Disposing of Atom Containers

Before you can add atoms to an atom container, you must first create the container by calling `QTNewAtomContainer`. The code sample shown in Listing 1-6 calls `QTNewAtomContainer` to create an atom container and then calls `SetSpriteData` to add data for a background sprite to the container.

Listing 1-6 Creating a new atom container

```
QTAtomContainer spriteData;
OSErr err
// create an atom container to hold a sprite's data
err=QTNewAtomContainer (&spriteData);
```

When you have finished using an atom container, you should dispose of it by calling the `QTDisposeAtomContainer` function. The sample code shown in Listing 1-7 calls `QTDisposeAtomContainer` to dispose of two atom containers, `sample` and `spriteData`.

Listing 1-7 Disposing of atom containers

```
if (sample)
    QTDisposeAtomContainer (sample);

if (spriteData)
    QTDisposeAtomContainer (spriteData);
```

Creating New Atoms

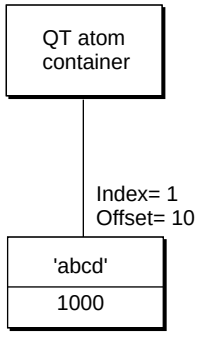
You use the `QTInsertChild` function to create new atoms and insert them in a QT atom container. The `QTInsertChild` function creates a new child atom for a parent atom or a leaf. The caller specifies an atom type and atom ID for the new atom. If you specify a value of 0 for the atom ID, `QTInsertChild` assigns a unique ID to the atom.

`QTInsertChild` inserts the atom in the parent's child list at the index specified by the `index` parameter; any existing atoms at the same index or greater are moved toward the end of the child list. If you specify a value of 0 for the `index` parameter, `QTInsertChild` inserts the atom at the end of the child list.

The following code sample creates a new QT atom container and calls `QTInsertChild` to add an atom. The resulting QT atom container is shown in Figure 1-6. The offset value 10 is returned in the `firstAtom` parameter.

```
QTAtom firstAtom;
QTAtomContainer container;
OSErr err
err=QTNewAtomContainer (&container);
if (!err)
    err=QTInsertChild (container, kParentAtomIsContainer, 'abcd',
        1000, 1, 0, nil, &firstAtom);
```

Figure 1-6 QT atom container after inserting an atom

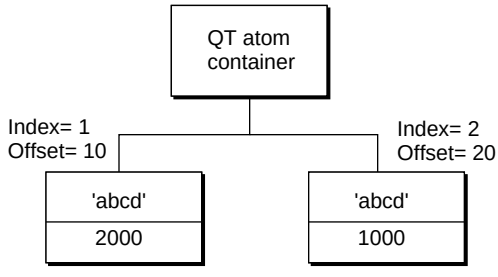


The following code sample calls `QTInsertChild` to create a second child atom. Because a value of 1 is specified for the `index` parameter, the second atom is inserted in front of the first atom in the child list; the index of the first atom is changed to 2. The offset value 10 is returned in the `secondAtom` parameter. The offset of the first atom is changed to 20. The resulting QT atom container is shown in Figure 1-7.

```
QTAtom secondAtom;

FailOSErr (QTInsertChild (container, kParentAtomIsContainer, 'abcd',
    2000, 1, 0, nil, &secondAtom));
```

Figure 1-7 QT atom container after inserting a second atom



You call the `QTFindChildByID` function to retrieve the changed offset of the first atom that was inserted, as shown in the following example. In this example, the `QTFindChildByID` function returns an offset of 20.

```
firstAtom = QTFindChildByID (container, kParentAtomIsContainer, 'abcd',
    1000, nil);
```

Listing 1-8 shows how the `QTInsertChild` function inserts a leaf atom into the atom container `sprite`. The new leaf atom contains a sprite image index as its data.

Listing 1-8 Inserting a child atom

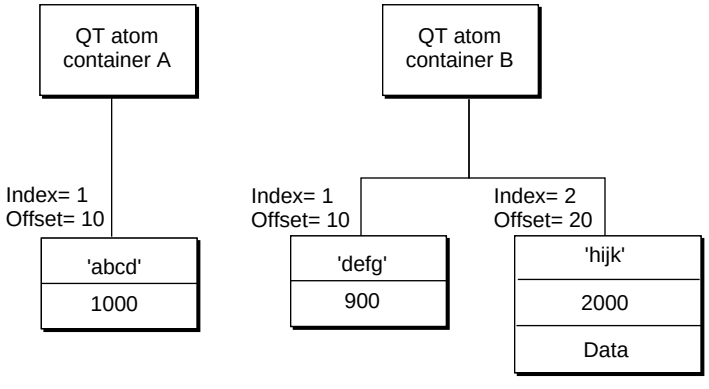
```
if ((propertyAtom = QTFindChildByIndex (sprite, kParentAtomIsContainer,
    kSpritePropertyImageIndex, 1, nil)) == 0)

    FailOSErr (QTInsertChild (sprite, kParentAtomIsContainer,
        kSpritePropertyImageIndex, 1, 1, sizeof(short),&imageIndex,
        nil));
```

Copying Existing Atoms

QuickTime provides several functions for copying existing atoms within an atom container. The `QTInsertChildren` function inserts a container of atoms as children of a parent atom in another atom container. Figure 1-8 shows two example QT atom containers, A and B.

Figure 1-8 Two QT atom containers, A and B



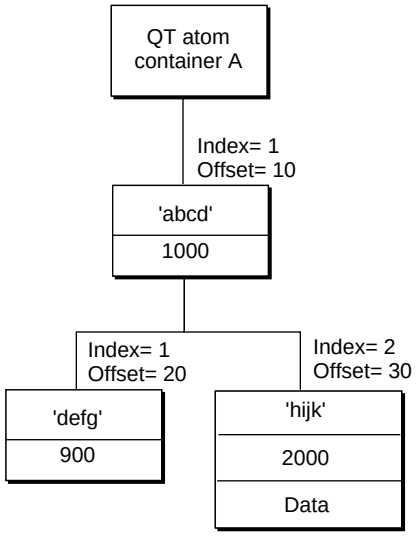
The following code sample calls `QTFindChildByID` to retrieve the offset of the atom in container A. Then, the code sample calls the `QTInsertChildren` function to insert the atoms in container B as children of the atom in container A. Figure 1-9 shows what container A looks like after the atoms from container B have been inserted.

```
QTAtom targetAtom;

targetAtom = QTFindChildByID (containerA, kParentAtomIsContainer, 'abcd',
    1000, nil);

FailOSErr (QTInsertChildren (containerA, targetAtom, containerB));
```

Figure 1-9 QT atom container after child atoms have been inserted



In Listing 1-9, the `QTInsertChild` function inserts a parent atom into the atom container `theSample`. Then, the sample code calls `QTInsertChildren` to insert the container `theSprite` into the container `theSample`. The parent atom is `newSpriteAtom`.

Listing 1-9 Inserting a container into another container

```
FailOSErr (QTInsertChild (theSample, kParentAtomIsContainer,
    kSpriteAtomType, spriteID, 0, 0, nil, &newSpriteAtom));

FailOSErr (QTInsertChildren (theSample, newSpriteAtom, theSprite));
```

QuickTime provides three other functions you can use to manipulate atoms in an atom container. The `QTReplaceAtom` function replaces an atom and its children with a different atom and its children. You can call the `QTSwapAtoms` function to swap the contents of two atoms in an atom container; after swapping, the ID and index of each atom remains the same. The `QTCopyAtom` function copies an atom and its children to a new atom container.

Retrieving Atoms From an Atom Container

QuickTime provides functions you can use to retrieve information about the types of a parent atom's children, to search for a specific atom, and to retrieve a leaf atom's data.

You can use the `QTCountChildrenOfType` and `QTGetNextChildType` functions to retrieve information about the types of an atom's children. The `QTCountChildrenOfType` function returns the number of children of a given atom type for a parent atom. The `QTGetNextChildType` function returns the next atom type in the child list of a parent atom.

You can use the `QTFindChildByIndex`, `QTFindChildByID`, and `QTNextChildAnyType` functions to retrieve an atom. You call the `QTFindChildByIndex` function to search for and retrieve a parent atom's child by its type and index within that type. Listing 1-10 shows the sample code function `SetSpriteData`, which updates an atom container that describes a sprite. For each property of the sprite that needs to be updated, `SetSpriteData` calls `QTFindChildByIndex` to retrieve the appropriate atom from the atom container. If the atom is found, `SetSpriteData` calls `QTSetAtomData` to replace the atom's data with the new value of the property. If the atom is not found, `SetSpriteData` calls `QTInsertChild` to add a new atom for the property.

Listing 1-10 Finding a child atom by index

```
OSErr SetSpriteData (QTAtomContainer sprite, Point *location,
    short *visible, short *layer, short *imageIndex)
{
    OSErr err = noErr;
    QTAtom propertyAtom;

    // if the sprite's visible property has a new value
    if (visible)
```

```
    {
        // retrieve the atom for the visible property
        // if none exists, insert one
        if ((propertyAtom = QTFindChildByIndex (sprite,
            kParentAtomIsContainer, kSpritePropertyVisible, 1,
            nil)) == 0)
            FailOSErr (QTInsertChild (sprite, kParentAtomIsContainer,
                kSpritePropertyVisible, 1, 1, sizeof(short), visible,
                nil))

        // if an atom does exist, update its data
        else
            FailOSErr (QTSetAtomData (sprite, propertyAtom,
                sizeof(short), visible));
    }

    // ...
    // handle other sprite properties
    // ...

    bail:
    return err;
}
```

You call the `QTFindChildByID` function to search for and retrieve a parent atom's child by its type and ID. The sample code function `AddSpriteToSample`, shown in Listing 1-11, adds a sprite, represented by an atom container, to a key sample, represented by another atom container. `AddSpriteToSample` calls `QTFindChildByID` to determine whether the atom container `theSample` contains an atom of type `kSpriteAtomType` with the ID `spriteID`. If not, `AddSpriteToSample` calls `QTInsertChild` to insert an atom with that type and ID. A value of 0 is passed for the `index` parameter to indicate that the atom should be inserted at the end of the child list. A value of 0 is passed for the `dataSize` parameter to indicate that the atom does not have any data. Then, `AddSpriteToSample` calls `QTInsertChildren` to insert the atoms in the container `theSprite` as children of the new atom. `FailIf` and `FailOSErr` are macros that exit the current function when an error occurs.

Listing 1-11 Finding a child atom by ID

```
OSErr AddSpriteToSample (QTAtomContainer theSample,
    QTAtomContainer theSprite, short spriteID)
{
    OSErr err = noErr;
    QTAtom newSpriteAtom;

    FailIf (QTFindChildByID (theSample, kParentAtomIsContainer,
        kSpriteAtomType, spriteID, nil), paramErr);

    FailOSErr (QTInsertChild (theSample, kParentAtomIsContainer,
        kSpriteAtomType, spriteID, 0, 0, nil, &newSpriteAtom));
    FailOSErr (QTInsertChildren (theSample, newSpriteAtom, theSprite));

bail:
    return err;
}
```

Once you have retrieved a child atom, you can call `QTNextChildAnyType` function to retrieve subsequent children of a parent atom. `QTNextChildAnyType` returns an offset to the next atom of any type in a parent atom's child list. This function is useful for iterating through a parent atom's children quickly.

QuickTime also provides functions for retrieving an atom's type, ID, and data. You can call `QTGetAtomTypeAndID` function to retrieve an atom's type and ID. You can access an atom's data in one of three ways.

- To copy an atom's data to a handle, you can use the `QTCopyAtomDataToHandle` function.
- To copy an atom's data to a pointer, you can use the `QTCopyAtomDataToPtr` function.
- To access an atom's data directly, you should lock the atom container in memory by calling `QTLockContainer`. Once the container is locked, you can call `QTGetAtomDataPtr` to retrieve a pointer to an atom's data. When you have finished accessing the atom's data, you should call the `QTUnlockContainer` function to unlock the container in memory.

Modifying Atoms

QuickTime provides functions that you can call to modify attributes or data associated with an atom in an atom container. To modify an atom's ID, you call the function `QTSetAtomID`.

You use the `QTSetAtomData` function to update the data associated with a leaf atom in an atom container. The `QTSetAtomData` function replaces a leaf atom's data with new data. The code sample in Listing 1-12 calls `QTFindChildByIndex` to determine whether an atom container contains a sprite's visible property. If so, the sample calls `QTSetAtomData` to replace the atom's data with a new visible property.

Listing 1-12 Modifying an atom's data

```
QTAtom propertyAtom;

// if the atom isn't in the container, add it
if ((propertyAtom = QTFindChildByIndex (sprite, kParentAtomIsContainer,
    kSpritePropertyVisible, 1, nil)) == 0)
    FailOSErr (QTInsertChild (sprite, kParentAtomIsContainer,
        kSpritePropertyVisible, 1, 0, sizeof(short), visible, nil))

// if the atom is in the container, replace its data
else
    FailOSErr (QTSetAtomData (sprite, propertyAtom, sizeof(short),
        visible));
```

Removing Atoms From an Atom Container

To remove atoms from an atom container, you can use the `QTRemoveAtom` and `QTRemoveChildren` functions. The `QTRemoveAtom` function removes an atom and its children, if any, from a container. The `QTRemoveChildren` function removes an atom's children from a container, but does not remove the atom itself. You can also use `QTRemoveChildren` to remove all the atoms in an atom container. To do so, you should pass the constant `kParentAtomIsContainer` for the `atom` parameter.

The code sample shown in Listing 1-13 adds override samples to a sprite track to animate the sprites in the sprite track. The `sample` and `spriteData` variables

are atom containers. The `spriteData` atom container contains atoms that describe a single sprite. The `sample` atom container contains atoms that describes an override sample.

Each iteration of the `for` loop calls `QTRemoveChildren` to remove all atoms from both the `sample` and the `spriteData` containers. The sample code updates the index of the image to be used for the sprite and the sprite's location and calls `SetSpriteData` (Listing 1-10), which adds the appropriate atoms to the `spriteData` atom container. Then, the sample code calls `AddSpriteToSample` (Listing 1-11) to add the `spriteData` atom container to the `sample` atom container. Finally, when all the sprites have been updated, the sample code calls `AddSpriteSampleToMedia` to add the override sample to the sprite track.

Listing 1-13 Removing atoms from a container

```
QTAtomContainer sample, spriteData;

// ...
// add the sprite key sample
// ...

// add override samples to make the sprites spin and move
for (i = 1; i <= kNumOverrideSamples; i++)
{
    QTRemoveChildren (sample, kParentAtomIsContainer);
    QTRemoveChildren (spriteData, kParentAtomIsContainer);

    // ...
    // update the sprite
    // - update the imageIndex
    // - update the location
    // ...

    // add atoms to spriteData atom container
    SetSpriteData (spriteData, &location, nil, nil, &imageIndex);

    // add the spriteData atom container to sample
    err = AddSpriteToSample (sample, spriteData, 2);

    // ...
```

```
    // update other sprites
    // ...

    // add the sample to the media
    err = AddSpriteSampleToMedia (newMedia, sample,
        kSpriteMediaFrameDuration, false);
}
```

Movie Toolbox Reference

This section describes the new and changed constants and functions in the Movie Toolbox.

Constants

This section describes the new constants provided by the Movie Toolbox.

Movie Exporting Flags

The `flags` parameter to the `ConvertMovieToFile` function specifies a set of movie conversion flags. QuickTime 2.0 provides these additional flags:

```
enum {
    showUserSettingsDialog  = 2,
    movieToFileOnlyExport   = 4,
    movieFileSpecValid      = 8
};
```

Flag descriptions

`showUserSettingsDialog`
: If this flag is set, the Save As dialog box will be displayed to allow the user to choose the type of file to export to, optional export settings, and the file name to export to.

`movieToFileOnlyExport`
: If this flag is set and the `showUserSettingsDialog` flag is set, the Save As dialog box restricts the user to those file

formats that are supported by movie data export components. If this flag is not set, the user will also be able to save the movie either as a self-contained movie or as a reference movie.

`movieFileSpecValid`

If this flag is set and the `showUserSettingsDialog` flag is set, the `name` field of the `outputFile` parameter is used as the default name of the exported file in the Save As dialog.

Movie Importing Flags

The `flags` parameter to the `ConvertFileToMovieFile` and `PasteHandleIntoMovie` functions specifiy a set of movie conversion flags. QuickTime 1.6.1 provides one additional flag:

```
enum {
    showUserSettingsDialog  = 2
};
```

Flag description

`showUserSettingsDialog`

If this flag is set, the user settings dialog box for that import operation can be displayed. For example, when importing a picture, this flag would display the Standard Compression dialog box so that the user could select the compression method.

Flattening Flags

The `flags` parameter to the `FlattenMovieData` function specifies a set of movie flattening flags. QuickTime 2.1 provides one new flag that you must set when specifying a data reference to flatten a movie to, instead of a file:

```
enum {
    flattenFSSpecPtrIsDataRefRecordPtr  = 1L << 4
};
```

Flag description

`flattenFSSpecPtrIsDataRefRecordPtr`
: Set this flag to 1 if the `FSSpec` pointer is a `DataReferencePtr`. This capability enables you to flatten movies to non-file system devices.

Interesting Times Flags

The `interestingTimeFlags` parameter to the interesting time functions (`GetMovieNextInterestingTime`, `GetTrackNexttInterestingTime`, and `GetMediaNextInterestingTime`) specifies a set of bit flags that specify search criteria. Normally, you use one of the interesting time functions to step forward to the next frame. These functions work well for most media types, including video and text. However, because QuickTime stores an entire MPEG stream as a single sample, stepping to the next sample skips to the end of the sequence. To solve this problem, QuickTime 2.1 introduced a new flag for the interesting time calls: `nextTimeStep`. This flag returns the time of the next frame, even if there are multiple frames per sample, for all media types including video, text, and MPEG. Applications which implement single stepping capabilities should always use this flag instead of `nextTimeMediaSample`.

```
enum {
    nextTimeStep    = 1 << 4
};
```

Flag description

`nextTimeStep`
: Searches for the next frame in the movie's media. Set this flag to 1 to step to the next frame.

Full Screen Flags

The `flags` parameter to the `BeginFullScreen` function specifies a set of bit flags that control certain aspects of the full-screen mode. QuickTime defines these constants that you can use in the `flags` parameter.

```
enum {
    fullScreenHideCursor            = 1L << 0,
    fullScreenAllowEvents           = 1L << 1,
```

```
    fullScreenDontChangeMenuBar     = 1L << 2,
    fullScreenPreflightSize         = 1L << 3
};
```

Flag description

`fullScreenHideCursor`

If this flag is set, `BeginFullScreen` hides the cursor. This is useful if you are going to play a QuickTime movie and do not want the cursor to be visible over the movie.

`fullScreenAllowEvents`

If this flag is set, your application intends to allow other applications to run (by calling `WaitNextEvent` to grant them processing time). In this case, `BeginFullScreen` does not change the monitor resolution, because other applications might depend on the current resolution.

`fullScreenDontChangeMenuBar`

If this flag is set, `BeginFullScreen` does not hide the menu bar. This is useful if you want to change the resolution of the monitor but still need to allow the user to access the menu bar.

`fullScreenPreflightSize`

If this flag is set, `BeginFullScreen` doesn't change any monitor settings, but returns the actual height and width that it would use if this bit were not set. This allows applications to test for the availability of a monitor setting without having to switch to it.

Text Sample Display Flags

The `displayflags` parameter to the `AddTESample` and `AddTextSample` functions control the behavior of the text media handler. QuickTime 2.5 provides these additional flags:

```
enum {
    dfContinuousScroll      = 1 << 9,
    dfFlowHoriz             = 1 << 10,
    dfContinuousKaraoke     = 1 << 11,
    dfDropShadow            = 1 << 12,
    dfAntiAlias             = 1 << 13,
    dfKeyedText             = 1 << 14,
```

```
    dfInverseHilite         = 1 << 15,
    dfTextColorHilite       = 1 << 16
};
```

Flag description

`dfContinuousScroll`

If this flag is set, the text media handler lets new samples cause previous samples to scroll out. You must also set `dfScrollIn` and/or `dfScrollOff` for this to take effect.

`dfFlowHoriz`

If this flag is set, the text media handler lets horizontally scrolled text flow within the text box instead of extending to the right.

`dfContinuousKaraoke`

If this flag is set, the text media handler highlights ignores the starting offset when highlighting text. Instead, it highlights text from the beginning of the text sample to the ending offset.

`dfDropShadow`

If this flag is set, the text media handler displays text with a drop shadow. When you use the `SetTextSampleData` function, the position and translucency of the drop shadow is under your application's control. For more information, see `SetTextSampleData` later in this chapter.

`dfAntiAlias`

If this flag is set, the text media handler displays text with anti-aliasing. Note that although anti-aliased text looks smoother, anti-aliasing can slow down performance.

`dfKeyedText`

If this flag is set, the text media handler renders text over the background without drawing the background color. This technique is also known as "masked text."

`dfInverseHilite`

If this flag is set, the text media handler highlights text using inverse video instead of the highlight color.

`dfTextColorHilite`

If this flag is set, the text media handler highlights text by changing the color of the text.

Modifier Input Types

The media input map describes the meaning of each input to a track. Each track has particular attributes such as size, position, and volume associated with it. The media input map of that track describes the mapping of track modifier inupts to track properties. When you want to modify the attributes of a track, you insert a track modifier input such as `kTrackModifierTypeMatrix` into the input map. The values stored in the modifier input you inserted will affect the values that are currently stored with the track.

Custom media handlers can define additional input types as necessary. Apple Computer reserves all input types consisting entirely of lower-case letters.

The following input types are currently defined:

```
enum {

    kTrackModifierTypeMatrix            = 1,
    kTrackModifierTypeClip              = 2,
    kTrackModifierTypeGraphicsMode      = 5,
    kTrackModifierTypeVolume            = 3,
    kTrackModifierTypeBalance           = 4,
    kTrackModifierTypeImage             = 'vide',
    kTrackModifierObjectMatrix          = 6,
    kTrackModifierObjectGraphicsMode    = 7,
    kTrackModifierType3d4x4Matrix       = 8,
    kTrackModifierCameraData            = 9
    kTrackModifierSoundLocalization     = 10
};
```

Constant descriptions

`kTrackModifierTypeMatrix`

Data sent to this input should be in the form of a QuickTime `MatrixRecord`. The matrix is concatenated with the track and movie matrices to determine the tracks final location and size. The matrix modifier describes relative, not absolute, position and scaling.

`kTrackModifierTypeClip`

Data sent to this input should be in the form of a QuickDraw region. The region is intersected with the track's source box. See *Inside Macintosh: QuickTime* for details on how a movie's tracks are composited together.

kTrackModifierTypeGraphicsMode

Data sent to this input should be in the form of a ModifierTrackGraphicsModeRecord data type. The contents of the record are used as the graphics mode setting for the track. The graphics mode is not combined with the track's current graphics mode, but rather overrides it . See "Data Types" later in this chapter for information about the ModifierTrackGraphicsModeRecord data structure.

kTrackModifierTypeVolume

Data sent to this input should be in the form of a 16-bit fixed-point number. This is the same format in which QuickTime sound volume levels are stored. The volume level is used as a scaling factor on the sound track's level. It is multiplied with the track and movie volumes to determine the track's overall volume.

kTrackModifierTypeBalance

Data sent to this input should be in the form of a 8-bit fixed-point number. This is the same format in which QuickTime balance values are stored. The balance value is used as the balance setting for the track. Unlike the volume modifier, it is not concatenated with the track's current balance level, but overrides the current balance level.

kTrackModifierTypeImage

Data sent to this input should be compressed video data, typically from a video track. This input type can be used with sprite and 3D tracks. For sprite tracks, the image data is used to replace the image of a specified image index in the sprite track. The index of the image to replace must be specified in the media input map when the reference is created. For 3D tracks, the image is used to provide a texture for the object specified in the input map.

kTrackModifierObjectMatrix

Data sent to this input should be in the form of a QuickTime MatrixRecord. The matrix is sent to a particular object within the receiving track, as specified by the kTrackModifierObjectID atom in the input map. The matrix acts as an override to the object's current matrix. For example, the matrix could be sent to a sprite within a sprite track. It would cause the sprite to move, not the entire sprite track as would kTrackModifierMatrix.

`kTrackModifierObjectGraphicsMode`

Data sent to this input should be in the form of a `ModifierTrackGraphicsModeRecord` data type. The contents of the record are used to vary the opacity of an object within the track. For example, you would use data sent to this input to vary the opacity of a sprite within a sprite track, rather than modifying the opacity of the entire sprite track. See "Data Types" later in this chapter for information about the `ModifierTrackGraphicsModeRecord` data structure.

`kTrackModifierType3d4x4Matrix`

Data sent to this input should be in the form of a QuickDraw 3D 4x4 matrix—`TQ3Matrix4x4`. This data is sent to objects within a QuickDraw 3D track. This matrix is concatenated with the all other matrices which effect the specified object. The structure is defined in the book *3D Graphics Programming with QuickDraw 3D.*

`kTrackModifierCameraData`

Data sent to this input should be in the form of a QuickDraw 3D Camera data structure—`TQ3CameraData`. This data is sent to a camera within a QuickDraw 3D track. The structure is defined in the book *3D Graphics Programming with QuickDraw 3D*.

`kTrackModifierSoundLocalization`

Data sent to this input should be in the form of a sound localization data record—`SSpLocalizationData`. This data overrides the sound localization settings already in use by the track.

Text Atom Types

The `dataType` parameter to the `AddTESample` and `AddTextSample` functions indicates the type of data in the handle. The following two types have been added:

```
enum {
    dropShadowOffsetType        = 'drpo',
    dropShadowTranslucencyType  = 'drpt'
};
```

Constant descriptions

`dropShadowOffsetType`
The drop shadow offset.

`dropShadowTranslucencyType`
The drop shadow translucency.

Background Sprites

You assign the following constant to a sprite's `kSpritePropertyLayer` property to designate the sprite as a background sprite.

```
enum {
    kBackgroundSpriteLayerNum   = 32767
};
```

Flags for Sprite Hit Testing

You can pass the following flags to the `SpriteWorldHitTest` function (page 1-122) and the `SpriteHitTest` function (page 1-126) to control sprite hit testing.

```
enum {
    spriteHitTestBounds = 1L << 0,
    spriteHitTestImage  = 1L << 1
};
```

Flag descriptions

`spriteHitTestBounds`
The specified location must be within the sprite's bounding box.

`spriteHitTestImage`
The specified location must be within the shape of the sprite's image.

Sprite Properties

The following constants represent the different properties of a sprite. When you call the `SetSpriteProperty` function (page 1-128) to set a sprite property, you pass one of these constants to specify the property you wish to modify.

```
enum {
    kSpritePropertyMatrix                   = 1,
    kSpritePropertyImageDescription         = 2,
    kSpritePropertyImageDataPtr             = 3,
    kSpritePropertyVisible                  = 4,
    kSpritePropertyLayer                    = 5,
    kSpritePropertyGraphicsMode             = 6,
    kSpritePropertyImageIndex               = 100
};
```

Constant descriptions

`kSpritePropertyMatrix`
: A matrix of type `MatrixRecord` that defines the sprite's display coordinate system.

`kSpritePropertyImageDescription`
: An image description handle that describes the sprite's image data.

`kSpritePropertyImageDataPtr`
: A pointer to the sprite's image data.

`kSpritePropertyVisible`
: A Boolean value that indicates whether the sprite is visible.

`kSpritePropertyLayer`
: A short integer value that defines the sprite's layer. You set this property to `kBackgroundSpriteLayerNum` to designate the sprite as a background sprite.

`kSpritePropertyGraphicsMode`
: A `ModifierTrackGraphicsModeRecord` value that specifies the graphics mode to be used when drawing the sprite.

`kSpritePropertyImageIndex`
: In a sprite track, the index of the sprite's image in the pool of shared images.

Flags for SpriteWorldIdle

You can pass the following flags as input to the `SpriteWorldIdle` function (page 1-120) to control drawing of the sprite world.

```
enum {
    kOnlyDrawToSpriteWorld  = 1L << 0,
    kSpriteWorldPreFlight   = 1L << 1
};
```

Flag descriptions

`kOnlyDrawToSpriteWorld`
: You set this flag to indicate that drawing should take place in the sprite world only; drawing to the final destination should be suppressed.

`kSpriteWorldPreFlight`
: You can set this flag to determine whether the sprite world has any invalid areas that need to be drawn. If so, the `SpriteWorldIdle` function returns the `kSpriteWorldNeedsToDraw` flag in the `flagsOut` parameter.

The following flags may be returned in the `flagsOut` parameter of the `SpriteWorldIdle` function.

```
enum {
    kSpriteWorldDidDraw         = 1L << 0,
    kSpriteWorldNeedsToDraw     = 1L << 1
};
```

Flag descriptions

`kSpriteWorldDidDraw`
: If set, this flag indicates that `SpriteWorldIdle` updated the sprite world.

`kSpriteWorldNeedsToDraw`
: If set, this flag indicates that the sprite world has invalid areas that need to be drawn.

Constants for QT Atom Functions

You can pass the `kParentAtomIsContainer` constant to QT atom functions that take an atom container and a parent atom as parameters. When passed in place of the parent atom, this constant indicates that the parent atom is the atom container itself.

```
enum {
    kParentAtomIsContainer  = 0
};
```

Data Types

This section describes new data structures provided by the Movie Toolbox.

Data Reference

To fully specify a data reference, it is necessary to provide the data reference itself, along with the type of the data reference (that is, the data reference handle does not contain the type of the data reference). The `DataReferenceRecord` data structure contains both of these pieces of information, making it possible to pass them to functions as a single parameter. The `FlattenMovieData` function uses the information in the data reference structure to flatten a movie to a data reference instead of to a file.

```
struct DataReferenceRecord {
    OSType  dataRefType;
    Handle  dataRef;
};

typedef struct DataReferenceRecord DataReferenceRecord;
typedef DataReferenceRecord *DataReferencePtr;
```

Field descriptions

`dataRefType` Specifies the type of data reference. For an alias data reference, you set the parameter to `rAliasType`, indicating that the reference is an alias. For a handle data reference, set the parameter to `HandleDataHandlerSubType`.

dataRef	Specifies the actual data reference. This parameter contains a handle to the information that identifies the file to be used. The type of information stored in the handle depends on the value of the dataRefType parameter. For example, if your application is loading the movie from a file, this parameter would contain an alias to the movie file.

Sample Reference

The SampleReferenceRecord structure is used to describe information about a sample or group of similar samples. This data structure is used by the GetMediaSampleReferences and AddMediaSampleReferences functions.

```
struct SampleReferenceRecord {
    long        dataOffset;
    long        dataSize;
    TimeValue   durationPerSample;
    long        numberOfSamples;
    short       sampleFlags;
};

typedef struct SampleReferenceRecord SampleReferenceRecord;
typedef SampleReferenceRecord *SampleReferencePtr;
```

Field descriptions

dataOffset	Specifies the offset into the movie data file. This field specifies the offset into the file of the sample data.
dataSize	Specifies the total number of bytes of sample data identified by the reference. All samples referenced by a single SampleReferenceRecord must be the same size.
durationPerSample	Specifies the duration of each sample in the reference. You must specify this parameter in the media's time scale. All samples referenced by a single SampleReferenceRecord must be the same duration.
numberOfSamples	Specifies the number of samples contained in the reference.
sampleFlag	Contains flags that control the operation. The following flag is available (set unused flags to 0):

`mediaSampleNotSync`
: Indicates that the sample to be added is not a sync sample. Set this flag to 1 if the sample is not an async sample. Set this flag to 0 if the sample is a sync sample.

Modifier Track Graphics Mode

The modifier track graphics mode structure contains information that defines the graphics mode setting for a track. Data in this structure indicates the graphics mode setting and the RGB op-color which is used with certain graphics modes. Data sent to the `kTrackModifierTypeGraphicsMode` input type should be in the form of a modifier track graphics mode structure.

```
struct ModifierTrackGraphicsModeRecord {
    long                    graphicsMode;
    RGBColor                opColor;
};
typedef struct ModifierTrackGraphicsModeRecord
ModifierTrackGraphicsModeRecord;
```

Field descriptions

`graphicsMode`
: Specifies the graphics mode setting.

`opColor`
: Contains an RGB color structure indicating the op-color to use with the graphics mode.

Sprite and Sprite World Identifiers

The sprite world and sprite data structures are private data structures. You identify a sprite world or a sprite data structure to the Sprite Toolbox by means of a data type that is supplied by the Sprite Toolbox. The following data types are currently defined:

`Sprite`
: Specifies the sprite for an operation. Your application obtains a sprite identifier when you create a new sprite by calling the `NewSprite` function (page 1-123).

`SpriteWorld`
: Specifies the sprite world for an operation. Your application obtains a sprite world identifier when you create a sprite world by calling the `NewSpriteWorld` function (page 1-117).

QT Atom

The `QTAtom` data type represents the offset of an atom within an atom container.

```
typedef long QTAtom;
```

QT Atom Type and ID

The `QTAtomType` data type represents the type of a QT atom. To be valid, a QT atom's type must have a non-zero value.

```
typedef long QTAtomType;
```

The `QTAtomID` data type represents the ID of a QT atom. To be valid, a QT atom's ID must have a non-zero value.

```
typedef long QTAtomID;
```

QT Atom Container

The `QTAtomContainer` data type is a handle to a QT atom container. Your application never modifies the contents of a QT atom container directly. Instead, you use the functions provided by QuickTime for creating and manipulating QT atom containers.

```
typedef Handle QTAtomContainer;
```

Functions for Getting and Playing Movies

The Movie Toolbox contains new and changed functions for getting and playing movies.

Movie Functions

NewMovieFromUserProc

The `NewMovieFromUserProc` function creates a movie from data that you provide. Your application defines a function that delivers the movie data to the Movie Toolbox. The Movie Toolbox calls your function, specifying the amount of data required and the location for the data.

```
pascal OSErr NewMovieFromUserProc (Movie *theMovie, short newMovieFlags,
                    Boolean *dataRefWasChanged, GetMovieUPP getProc,
                    void *refCon, Handle defaultDataRef,
                    OSType dataRefType);
```

`theMovie` Contains a pointer to a field that is to receive the new movie's identifier. If the function cannot load the movie, the returned identifier is set to `nil`.

`newMovieFlags`

Controls the operation of the `NewMovieFromUserProc` function. The following flags are valid (be sure to set unused flags to 0):

`newMovieActive`

Controls whether the new movie is active. Set this flag to 1 to make the new movie active. You can make a movie active or inactive by calling the `SetMovieActive` function.

`newMovieDontResolveDataRefs`

Controls how completely the Movie Toolbox resolves data references in the movie resource. If you set this flag to 0, the toolbox tries to completely resolve all data references in the resource. This may involve searching for files on remote volumes. If you set this flag to 1, the Movie Toolbox only looks in the specified data reference.

If the Movie Toolbox cannot completely resolve all the data references, it still returns a valid movie identifier. In this case, the Movie Toolbox also sets the current error value to `couldNotResolveDataRef`.

`newMovieDontAskUnresolvedDataRefs`
Controls whether the Movie Toolbox asks the user to locate files. If you set this flag to 0, the Movie Toolbox asks the user to locate files that it cannot find on available volumes. If the Movie Toolbox cannot locate a file even with the user's help, the function returns a valid movie identifier and sets the current error value to `couldNotResolveDataRef`.

`newMovieDontAutoAlternate`
Controls whether the Movie Toolbox automatically selects enabled tracks from alternate track groups. If you set this flag to 1, the Movie Toolbox does not automatically select tracks for the movie—you must enable and disable tracks yourself.

`dataRefWasChanged`
Contains a pointer to a Boolean value. The Movie Toolbox sets the Boolean to indicate whether it had to change any data references while resolving them. The toolbox sets the Boolean value to true if any references were changed. Use the `UpdateMovieResource` function to preserve these changes.

Set the `dataRefWasChanged` parameter to `nil` if you do not want to receive this information.

`getProc` Contains a pointer to a function in your application. This function is responsible for providing the movie data to the Movie Toolbox.

`refCon` Contains a reference constant (defined as a void pointer). The Movie Toolbox provides this value to the function identified by the `getProc` parameter.

`defaultDataRef`
: Specifies the default data reference. This parameter contains a handle to the information that identifies the file to be used to resolve any data references and as a starting point for any Alias Manager searches.

  The type of information stored in the handle depends upon the value of the `dataRefType` parameter. For example, if your application is loading the movie from a file, you would refer to the file's alias in the `defaultDataRef` parameter, and set the `dataRefType` parameter to `rAliasType`.

  If you do not want to identify a default data reference, set the parameter to `nil`.

`dataRefType`
: Specifies the type of data reference. If the data reference is an alias, you must set the parameter to `rAliasType`, indicating that the reference is an alias.

DESCRIPTION

Your application must define a function that provides the movie data to the Movie Toolbox. You specify that function to the Movie Toolbox with the `getProc` parameter. That function must support the following interface:

```
pascal OSErr MyGetMovieProc (long offset, long size, void *dataPtr,
                    void *refCon);
```

`offset`
: Specifies the offset into the movie resource (not the movie file). This is the location from which your function retrieves the movie data.

`size`
: Specifies the amount of data requested by the Movie Toolbox, in bytes.

`dataPtr`
: Specifies the destination for the movie data.

`refCon`
: Contains a reference constant (defined as a void pointer). This is the same value you provided to the Movie Toolbox when you called the `NewMovieFromUserProc` function.

Normally, when a movie is loaded from a file (say, by means of the `NewMovieFromFile` function), the Movie Toolbox uses that file as the default data reference. Since the `NewMovieFromUserProc` function does not require a file

specification, your application should specify the file to be used as the default data reference using the `defaultDataRef` and `dataRefType` parameters.

SPECIAL CONSIDERATIONS

The Movie Toolbox automatically sets the movie's graphics world based upon the current graphics port. Be sure that your application's graphics world is valid before you call this function.

RESULT CODES

`paramErr`	–50	Invalid parameter specified
`noMovieFound`	–2048	Toolbox cannot find a movie in the movie file

Memory Manager errors
Resource Manager errors

NewMovieFromFile

The `NewMovieFromFile` function (QuickTime 2.0 or higher) works with some files that do not contain movie resources. In some cases, the data in a file is already sufficiently well-formatted for QuickTime or its components to understand. For example, the AIFF movie data import component can understand AIFF sound files and import the sound data into a QuickTime movie. When the `NewMovieFromFile` function encounters a file that does not contain a movie resource, the function tries to find a movie import component that can understand the data and create a movie. For more information about new capabilities of movie data import components, see the chapter "Movie Data Exchange Components" elsewhere in this document. This also works for MPEG, µLaw (`.AU`), and Wave (`.WAV`) file types.

NewMovieFromDataRef

Use the new `NewMovieFromDataRef` function to create a movie from any device with a corresponding data handler. You are no longer restricted to instantiating

a movie from a file stored on an HFS volume. With this new function, you can now instantiate a movie from any device.

```
pascal OSErr NewMovieFromDataRef (
                    Movie *theMovie,
                    short flags,
                    short *id,
                    Handle dataRef,
                    OSType dataRefType);
```

`theMovie` Contains a pointer to a field that is to receive the new movie's identifier. If the function cannot load the movie, the returned identifier is set to `nil`.

`flags` Controls the operation of the `NewMovieFromDataRef` function. The following flags are valid (be sure to set unused flags to 0):

`newMovieActive`
Controls whether the new movie is active. Set this flag to 1 to make the new movie active. You can make a movie active or inactive by calling the `SetMovieActive` function.

`newMovieDontResolveDataRefs`
Controls how completely the Movie Toolbox resolves data references in the movie resource. If you set this flag to 0, the toolbox tries to completely resolve all data references in the resource. This may involve searching for files on remote volumes. If you set this flag to 1, the Movie Toolbox only looks in the specified data reference.

If the Movie Toolbox cannot completely resolve all the data references, it still returns a valid movie identifier. In this case, the Movie Toolbox also sets the current error value to `couldNotResolveDataRef`.

`newMovieDontAskUnresolvedDataRefs`
Controls whether the Movie Toolbox asks the user to locate files. If you set this flag to 0, the Movie Toolbox asks the user to locate files that it cannot find on available volumes. If the Movie

Toolbox cannot locate a file even with the user's help, the function returns a valid movie identifier and sets the current error value to `couldNotResolveDataRef`.

`newMovieDontAutoAlternate`
Controls whether the Movie Toolbox automatically selects enabled tracks from alternate track groups. If you set this flag to 1, the Movie Toolbox does not automatically select tracks for the movie—you must enable and disable tracks yourself.

`id` Contains a pointer to the field that specifies the resource containing the movie data that is to be loaded. If the field referred to by the `id` parameter is set to 0, the Movie Toolbox loads the first movie resource it finds in the specified file. The toolbox then returns the movie's resource ID number in the field referred to by the `id` parameter. The following enumerated constant is available:

`movieInDataForkResID`
Indicates the movie was loaded from the data fork. If the resource was stored in the file's data fork, the Movie Toolbox sets the returned value to `movieInDataForkResID(-1)`. In this case, you cannot add a movie resource to the file unless you create a resource fork in the movie file.

If the `id` parameter is set to `nil`, the Movie Toolbox loads the first movie resource it finds in the specified file and does not return that resource 's ID number.

`dataRef` Specifies the default data reference. This parameter contains a handle to the information that identifies the file to be used to resolve any data references and as a starting point for any Alias Manager searches.

The type of information stored in the handle depends upon the value of the `dataRefType` parameter. For example, if your application is loading the movie from a file, you would refer to the file's alias in the `DataRef` parameter, and set the `dataRefType` parameter to `rAliasType`.

If you do not want to identify a default data reference, set the parameter to `nil`.

`dataRefType` Specifies the type of data reference. If the data reference is an alias, you must set the parameter to `rAliasType`, indicating that the reference is an alias.

DISCUSSION

`NewMovieFromDataRef` is intended for use by specialized applications that need to instantiate movies from devices not visible to the file system. Most applications should continue to use `NewMovieFromFile`.

RESULT CODES

`badImageDescription`	–2001	Problem with an image description
`badPublicMovieAtom`	–2002	Movie file corrupted
`cantFindHandler`	–2003	Cannot locate a handler
`cantOpenHandler`	–2004	Cannot open a handler

File Manager errors
Memory Manager errors
Resource Manager error

ConvertFileToMovieFile

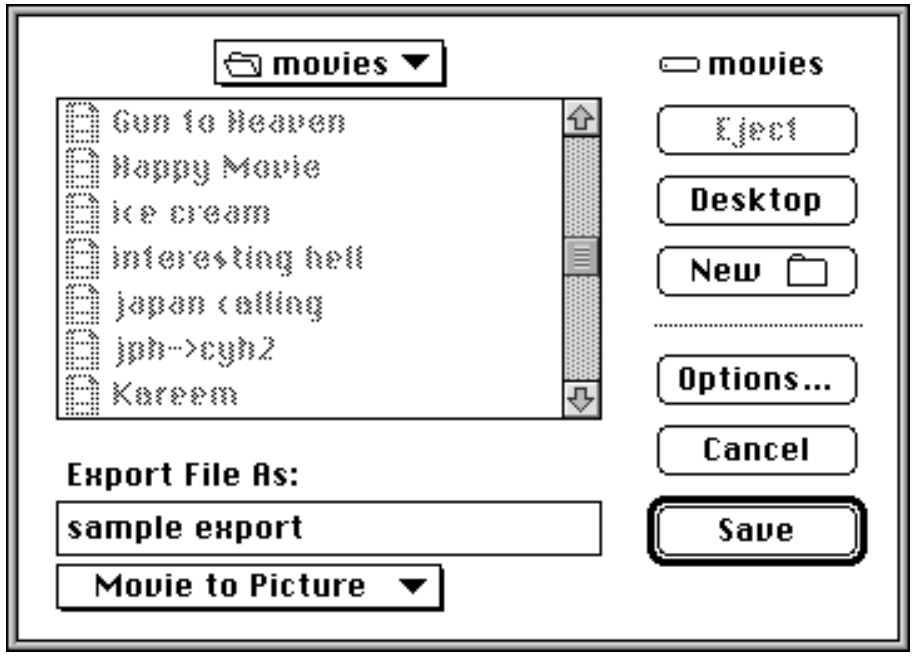
As of QuickTime 1.6.1, the `ConvertFileToMovieFile` function supported a user settings dialog box for import operations. Your application controls whether this dialog appears by setting the value of the `flags` parameter in the `ConvertFileToMovieFile` function. This function supports the following new flag:

`showUserSettingsDialog`
Controls whether the user settings dialog for the specified import operation can appear. Set this flag to 1 to display the user settings dialog.

ConvertMovieToFile

The `ConvertMovieToFile` function now supports a "Save As..." dialog box. The dialog allows the user to specify the file name and type. Supported types include standard QuickTime movies, flattened movies, single-fork flattened movies, and any format that is supported by a movie data export component. Figure 1 shows a sample "Save As..." dialog box.

Figure 1-10 Sample "Save As..." dialog box



Your application controls whether this dialog appears by setting the value of the `flags` parameter to the `ConvertMovieToFile` function. The function supports the following flags:

showUserSettingsDialog
: If this bit is set, the Save As dialog box will be displayed to allow the user to choose the type of file to export to, as well as the file name to export to.

movieToFileOnlyExport
: If this bit is set and the showUserSettingsDialog bit is set, the Save As dialog box restricts the user to those file formats that are supported by movie data export components.

movieFileSpecValid
: If this bit is set and the showUserSettingsDialog bit is set, the name field of the outputFile parameter is used as the default name of the exported file in the Save As dialog.

The following code shows how to call this function to provide a simple export capability.

```
err = ConvertMovieToFile (theMovie,     /* identifies movie */
                    nil,                /* all tracks */
                    nil,                /* no output file */
                    0,                  /* no file type */
                    0,                  /* no creator */
                    -1,                 /* script */
                    nil,                /* no resource ID */
                    createMovieFileDeleteCurFile |
                        showUserSettingsDialog |
                        movieToFileOnlyExport,
                    0);                 /* no specific component */
```

FlattenMovie and FlattenMovieData

The Movie Toolbox, through the new SetTrackLoadSettings function, now allows you to set a movie's preloading guidelines when you create the movie. The preload information is preserved when you save or flatten the movie (using either the FlattenMovie or FlattenMovieData functions). In flattened movies, the tracks that are to be preloaded are stored at the start of the movie, rather than being interleaved with the rest of the movie data. This greatly improves preload performance because it is not necessary for the device storing the movie data to seek during retrieval of the data to be preloaded.

For more information about preloading, see the discussion of the `SetTrackLoadSettings` function in "Enhancing Movie Playback Performance."

Instead of flattening to a file, you can now also specify a data reference to flatten a movie to, instead of a file. The `FSSpec` parameter to the `FlattenMovieData` function usually contains a pointer to the file system specification for the movie file to be created. In place of the `FSSpec` parameter, QuickTime 2.1 enables you to pass a pointer to a data reference structure. The data reference structure defines the data reference to flatten the movie data to. For more information about this structure, see "Data Types" earlier in this chapter.

Enhancing Movie Playback Performance

Two new functions allow you to get and set a portion of a preloaded track. There is also a new function for working with modifier tracks.

SetTrackLoadSettings

The `SetTrackLoadSettings` function allows you to specify a portion of a track that is to be loaded into memory whenever it is played.

```
pascal void SetTrackLoadSettings (Track theTrack, TimeValue preloadTime,
                    TimeValue preloadDuration, long preloadFlags,
                    long defaultHints);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack`.

`preloadTime` Specifies the starting point of the portion of the track to be preloaded. Set this parameter to –1 if you want to preload the entire track (in this case the function ignores the `preloadDuration` parameter). This should be specified in the movie's time scale.

`preloadDuration`
Specifies the amount of the track to be preloaded, starting from the time specified in the `preloadTime` parameter. If you are preloading the entire track, the function ignores this parameter.

`preloadFlags` Controls when the Movie Toolbox preloads the track. The function supports the following flag values:

`preloadAlways` Specifies that the Movie Toolbox should always preload this track, even if the track is disabled.

`preloadOnlyIfEnabled`
Specifies that the Movie Toolbox should preload this track only when the track is enabled.

Set this parameter to 0 if you do not want to preload the track.

`defaultHints` Specifies playback hints for the track. You may specify any of the supported hints flags. See "Hints," earlier in this chapter, for some flags that are new with QuickTime 2.0.

DESCRIPTION

The `SetTrackLoadSettings` allows you to control how the Movie Toolbox preloads the tracks in your movie. By using these settings, you make this information part of the movie, so that the preloading takes place every time the movie is opened, without an application having to call the `LoadTrackIntoRAM` function. Consequently, you should use this feature carefully, so that your movies do not consume large amounts of memory when opened.

SPECIAL CONSIERATIONS

The Movie Toolbox transfers this preload information when you call the `CopyTrackSettings` function. In addition, the preload information is preserved when you save or flatten a movie (using either the `FlattenMovie` or `FlattenMovieData` functions). In flattened movies, the tracks that are to be preloaded are stored at the start of the movie, rather than being interleaved with the rest of the movie data. This improves preload performance.

RESULT CODES

invalidTrack	–2009	This track is corrupted or invalid

GetTrackLoadSettings

The GetTrackLoadSettings function allows you to retrieve a track's preload information.

```
pascal void GetTrackLoadSettings (Track theTrack,
                    TimeValue *preloadTime, TimeValue *preloadDuration,
                    long *preloadFlags, long *defaultHints);
```

theTrack Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as NewMovieTrack and GetMovieTrack.

preloadTime Specifies a field to receive the starting point of the portion of the track to be preloaded. The Movie Toolbox returns a value of –1 if the entire track is to be preloaded.

preloadDuration
Specifies a field to receive the amount of the track to be preloaded, starting from the time specified in the preloadTime parameter. If the entire track is to be preloaded, this value is meaningless.

preloadFlags Specifies a field to receive the flags that control when the Movie Toolbox preloads the track. The function supports the following flag values:

preloadAlways Specifies that the Movie Toolbox always preloads this track.

preloadOnlyIfEnabled
Specifies that the Movie Toolbox preloads this track only when the track is enabled.

defaultHints Specifies a field to receive the playback hints for the track.

RESULT CODES

invalidTrack	–2009	This track is corrupted or invalid

GetTrackDisplayMatrix

The GetTrackDisplayMatrix function returns a matrix which is the concatenation of all matrices currently effecting the track's location, scaling, and so on. This includes the movie's matrix, the track's matrix, and the modifier matrix. Since modifier information is passed between tracks at MoviesTask time, the information returned by this call will represent the matrix in effect at the last MoviesTask call.

```
pascal OSErr GetTrackDisplayMatrix(
                    Track theTrack,
                    MatrixRecord *matrix );
```

theTrack — Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as NewMovieTrack and GetMovieTrack.

matrix — Contains a pointer to a matrix structure.

Note
To determine the entire clip of a track at the current time using GetTrackDisplayBoundsRgn. The results of GetTrackDisplayBoundsRgn take into account any clip regions provided by modifier tracks.

RESULT CODES

invalidTrack	–2009	This track is corrupted or invalid

Generating Pictures From Movies

When memory is low, the GetMoviePict function now reports out of memory errors instead of returning empty pictures.

Working with Progress and Cover Functions

SetMovieDrawingCompleteProc

The `SetMovieDrawingCompleteProc` function allows you to assign a drawing-complete function to a movie. The Movie Toolbox calls this function based upon guidelines you establish when you assign the function to the movie.

```
pascal void SetMovieDrawingCompleteProc (Movie theMovie, long flags,
                    MovieDrawingCompleteProcPtr proc, long refCon);
```

`theMovie` Specifies the movie for this operation. Your application obtains this identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle`.

`flags` Contains information that controls when your drawing complete function is called. The following values are supported:

`movieDrawingCallWhenChanged`
Specifies that the Movie Toolbox should call your drawing-complete function only when the movie has changed.

`movieDrawingCallAlways`
Specifies that the Movie Toolbox should call your drawing-complete function every time your application calls the `MoviesTask` function.

`proc` Contains a pointer to your drawing-complete function. Set this parameter to `nil` if you want to remove your function.

`refCon` Contains a value that the Movie Toolbox provides to your drawing-complete function.

DESCRIPTION

Your drawing-complete function must support the following interface:

```
typedef pascal OSErr (*MovieDrawingCompleteProcPtr)(Movie theMovie, long
                    refCon);
```

theMovie	Specifies the movie for this operation.
refCon	Contains the reference constant you supplied when your application called the SetMovieDrawingCompleteProc function.

Note
Some media handlers may take less efficient playback paths when a drawing complete proc is used. This function should only be used when absolutely necessary.

RESULT CODES

invalidMovie	–2010	This movie is corrupted or invalid

SetMovieCoverProcs

If a movie with semi-transparent tracks has a movie uncover procedure (set with the SetMovieCoverProcs function), the uncover procedure is now called before each frame to fill or erase the background. Before QuickTime 1.6.1, the Movie Toolbox performed the erase, which limited a cover procedure-aware application's options.

GetMovieCoverProcs

The GetMovieCoverProcs function allows you to retrieve the cover functions that you set with the SetMovieCoverProcs function.

```
pascal OSErr GetMovieCoverProcs(
                    Movie theMovie,
                    MovieRgnCoverUPP *uncoverProc,
                    MovieRgnCoverUPP *coverProc,
                    long *refcon)
```

theMovie	Specifies the movie for this operation. Your application obtains this identifier from such functions as NewMovie, NewMovieFromFile, and NewMovieFromHandle.

uncoverProc	Where to return the current uncover procedure. This value is set to nil if no uncover procedure was specified.
coverProc	Where to return the current cover procedure. This value is set to nil if no cover prodecure was specified.
refcon	Specifies a reference constant. The Movie Toolbox passes this value to your cover functions.

DISCUSSION

The GetMovieCoverProcs function returns the uncover and cover functions for the movie as well as the reference constant for the cover functions.

RESULT CODES

invalidMovie	–2010	This movie is corrupted or invalid

Functions That Modify Movie Properties

Working With Movie Spatial Characteristics

SetMovieColorTable

The SetMovieColorTable function allows you to associate a color table with a movie.

```
pascal OSErr SetMovieColorTable (Movie theMovie, CTabHandle ctab);
```

theMovie	Specifies the movie for this operation. Your application obtains this identifier from such functions as NewMovie, NewMovieFromFile, and NewMovieFromHandle.
ctab	Contains a handle to the color table. Set this parameter to nil to remove the movie's color table.

DESCRIPTION

The color table you supply may be used to modify the palette of indexed display devices at playback time. If you are using the movie controller, be sure to set the `mcFlagsUseWindowPalette` flag. If you are not using the movie controller, you should retrieve the movie's color table (using the `GetMovieColorTable` function) and supply it to the Palette Manager.

The Movie Toolbox makes a copy of the color table, so it is your responsibility to dispose of the color table when you are done with it. If the movie already has a color table, the Movie Toolbox uses the new table to replace the old one.

The `CopyMovieSettings` function copies the movie's color table, along with the other settings information.

RESULT CODES

`invalidMovie`	–2010	The movie is corrupted or invalid

Memory Manager errors

GetMovieColorTable

The `GetMovieColorTable` function allows you to retrieve a movie's color table.

```
pascal OSErr GetMovieColorTable (Movie theMovie, CTabHandle *ctab);
```

`theMovie` Specifies the movie for this operation. Your application obtains this identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle`.

`ctab` Contains a pointer to a field that is to receive a handle to the movie's color table. If the movie does not have a color table, the Movie Toolbox sets the field to `nil`.

DESCRIPTION

The Movie Toolbox returns a copy of the color table, so it is your responsibility to dispose of the color table when you are done with it.

RESULT CODES

invalidMovie	–2010	The movie is corrupted or invalid

Memory Manager errors

SetTrackGWorld

The SetTrackGWorld function allows you to force a track to draw into a particular GWorld. This GWorld may be different from that of the movie.

```
pascal void SetTrackGWorld(
                    Track theTrack,
                    CGrafPtr port,
                    GDHandle gdh,
                    TrackTransferUPP proc,
                    long refCon)
```

theTrack	Specifies the track for this operation. Your application obtains this identifier from such functions as GetMovieTrack, GetMovieIndTrack, and GetMovieIndTrackType.
port	Points to the graphics port structure or graphics world to which to draw the track. Set this parameter to nil to use the movie's graphics port.
gdh	Contains a handle to the movie's graphics device structure. Set this parameter to nil to use the current device. If the port parameter specifies a graphics world, set this parameter to nil to use that graphics world's graphics device.
proc	Contains a pointer to your transfer procedure. Set this parameter to nil if you want to remove your transfer procedure.
refCon	Contains a value to pass to your transfer procedure.

DISCUSSION

After the SetTrackGWorld function draws a track, it calls your transfer procedure to copy the track to the actual movie GWorld. When your transfer procedure is called, the current GWorld is set to the correct destination. You can also install a transfer procedure and set the GWorld to nil. Setting GWorld to

nil calls your transfer procedure only as a notification that the track has been drawn; no transfer needs to take place.

RESULT CODES

invalidTrack	–2009	This track is corrupted or invalid

Locating a Movie's Tracks and Media Structures

GetMovieIndTrackType

The GetMovieIndTrackType function allows you to search for all of a movie's tracks that share a given media type or media characteristic.

```
pascal Track GetMovieIndTrackType (Movie theMovie, long index, OSType
                    trackType, long flags);
```

theMovie	Specifies the movie for this operation. Your application obtains this identifier from such functions as NewMovie, NewMovieFromFile, and NewMovieFromHandle.
index	Specifies the index value of the track for this operation. This is not that same as the track's index value in the movie. Rather, this parameter is an index into the set of tracks that meet your other selection criteria.
trackType	Contains either a media type or a media characteristic value. The Movie Toolbox applies this value to the search, and returns information about tracks that meet this criterion. You indicate whether you have specified a media type or characteristic value by setting the flags parameter appropriately.
flags	Contains flags that control the search operation. The following flags are valid (note that you may not set both movieTrackMediaType and movieTrackCharacteristic to 1):

`movieTrackMediaType`
: Indicates that the `trackType` parameter contains a media type value. Set this flag to 1 if you are supplying a media type value (such as `VideoMediaType`).

`movieTrackCharacteristic`
: Indicates that the `trackType` parameter contains a media characteristic value. Set this flag to 1 if you are supplying a media characteristic value (such as `VisualMediaCharacteristic`).

`movieTrackEnabledOnly`
: Specifies that the Movie Toolbox should only search enabled tracks. Set this track to 1 to limit the search to enabled tracks.

function result Returns the track identifier of your selected track or `nil`.

DESCRIPTION

The Movie Toolbox returns the track identifier that corresponds to the track that meets your selection criteria. If the Movie Toolbox cannot find a matching track, in returns a value of `nil`.

Note that the `index` parameter does not work the same way that is does in the `GetMovieIndTrack` function. With the `GetMovieIndTrackType` function, the index parameter specifies an index into the set of tracks that meet your other selection criteria. For example, in order to find the third track that supports the sound characteristic, you would call the function in the following manner:

```
theTrack = GetMovieIndTrackType (theMovie,
                    3,
                    AudioMediaCharacteristic,
                    movieTrackCharacteristic);
```

RESULT CODES

paramErr	–50	Invalid parameter specified
invalidMovie	–2010	The movie is corrupted or invalid

Working With Track References

Track references allow you to relate tracks to one another. For example, this can be useful to identify the text track that contains the subtitles for a movie's audio track, and relating the text track to a particular audio track. See "Track References", earlier in this chapter, for more information about track references.

The AddTrackReference function allows you to relate one track to another. The DeleteTrackReference function removes that relationship. The SetTrackReference and GetTrackReference functions allow you to modify an existing track reference so that it identifies a different track. The GetNextTrackReferenceType and GetTrackReferenceCount functions allow you to scan all of a track's track references.

AddTrackReference

The AddTrackReference function allows you to add a new track reference to a track.

```
pascal OSErr AddTrackReference (Track theTrack, Track refTrack,
                    OSType refType, long *addedIndex);
```

theTrack	Identifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as NewMovieTrack and GetMovieTrack.
refTrack	Specifies the track to be identified in the track reference.
refType	Specifies the type of reference.
addedIndex	Contains a pointer to a long. The Movie Toolbox returns the index value assigned to the new track reference. If you do not want this information, set this parameter to nil.

RESULT CODES

`invalidTrack`	–2009	This track is corrupted or invalid

Memory Manager errors

DeleteTrackReference

The `DeleteTrackReference` function allows you to remove a track reference from a track.

```
pascal OSErr DeleteTrackReference (Track theTrack, OSType refType, long
                    index);
```

`theTrack` Identifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack`.

`refType` Specifies the type of reference.

`index` Specifies the index value of the reference to be deleted. You obtain this index value when you create the track reference.

DESCRIPTION

This function deletes a track reference from a track. If there are additional track references with higher index values, the Movie Toolbox automatically renumbers those references, decrementing their index values by 1.

RESULT CODES

`paramErr`	–50	Invalid parameter specified
`invalidTrack`	–2009	This track is corrupted or invalid

Memory Manager errors

SetTrackReference

The `SetTrackReference` function allows you to modify an existing track reference. You may change the track reference so that it identifies a different track in the movie.

```
extern pascal OSErr SetTrackReference (Track theTrack, Track refTrack,
                    OSType refType, long index);
```

`theTrack` Identifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack`.

`refTrack` Specifies the track to be identified in the track reference. The Movie Toolbox uses this information to update the existing track reference.

`refType` Specifies the type of reference.

`index` Specifies the index value of the reference to be changed. You obtain this index value when you create the track reference.

RESULT CODES

`paramErr`	–50	Invalid parameter specified
`invalidTrack`	–2009	This track is corrupted or invalid

GetTrackReference

The `GetTrackReference` function allows you to retrieve the track identifier contained in an existing track reference.

```
pascal Track GetTrackReference (Track theTrack, OSType refType, long
                    index);
```

`theTrack` Identifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack`.

`refType` Specifies the type of reference.

`index`	Specifies the index value of the reference found. You obtain this index value when you create the track reference.
function result	Returns the track identifier contained in the specified track reference.

DESCRIPTION

This function returns the track identifier that is contained in the specified track reference. If the Movie Toolbox cannot locate the track reference corresponding to your specifications, it returns a value of `nil`.

GetNextTrackReferenceType

The `GetNextTrackReferenceType` function allows you to determine all of the track reference types that are defined for a given track.

```
pascal OSType GetNextTrackReferenceType (Track theTrack, OSType refType);
```

`theTrack`	Identifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack`.
`refType`	Specifies the type of reference. Set this parameter to 0 to retrieve the first track reference type. On subsequent requests, use the previous value returned by this function.
function result	Returns an operating-system data type.

DESCRIPTION

This function returns an operating-system data type containing the next track reference type value defined for the track. There is no implied ordering of the returned values. When you reach the end of the track's reference types, this function sets the returned value to 0. You can use this value to stop your scanning loop.

GetTrackReferenceCount

The `GetTrackReferenceCount` function allows you to determine how many track references of a given type exist for a track.

```
pascal long GetTrackReferenceCount (Track theTrack, OSType refType);
```

`theTrack` Identifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack`.

`refType` Specifies the type of reference. The Movie Toolbox determines the number of track references of this type.

function result Returns a long integer or 0.

DESCRIPTION

This function returns long integer that contains the number of track references of the specified type in the track. If there are no references of the type you have specified, the function returns a value of 0.

Working With Sound

The following calls operate on the static 3D Sound Setting for a track. By constantly setting the value it is possible for an application to make a track's sound move in 3D space. If it is necessary to store dynamically changing 3D Sound settings for the track, this can be done using the Modifier Track mechanism in conjunction with a Tween Track. This is described below.

SetTrackSoundLocalizationSettings

`SetTrackSoundLocalizationSettings` replaces the current 3D Sound settings for the specified track with the new `SSpLocalizationData` record contained in the settings handle. The effect of the new 3D Sound setting will take place immediately. This call will aways store the new record passed, even if the track or the computer is not capable of actually meeting the request. You can pass a

nil handle to indicate that no 3D Sound effects should be used for this track. When the movie is saved, the 3D Sound settings is saved with it.

SetTrackSoundLocalizationSettings makes a copy of the handle passed, so the caller is responsible for disposing of the settings handle.

```
pascal OSErr SetTrackSoundLocalizationSettings (Track theTrack, Handle
                    settings)
```

theTrack	Identifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as NewMovieTrack and GetMovieTrack.
settings	The settings you want to apply, in the format of a Sound Sprockets SSpLocalizationData record.

The following example source code shows how to set the static 3D Sound Setting for a track using SetTrackSoundLocalizationSettings.

```
void setTrackSoundLocalization(Track t)
{
    SSpLocalizationData loc;
    Handle h;
    OSErr err;

    loc.cpuLoad = 0;
    loc.medium = kSSpMedium_Air;
    loc.humidity = 0;
    loc.roomSize = 250;
    loc.roomReflectivity = -5;
    loc.reverbAttenuation = -5;
    loc.sourceMode = kSSpSourceMode_Localized;
    loc.referenceDistance = 1;
    loc.coneAngleCos = 0;
    loc.coneAttenuation = 0;
    loc.currentLocation.elevation = 0;
    loc.currentLocation.azimuth = 0;
    loc.currentLocation.distance = 2;
    loc.currentLocation.projectionAngle = 0;
    loc.currentLocation.sourceVelocity = 0;
    loc.currentLocation.listenerVelocity = 0;
    loc.reserved0 = 0;
    loc.reserved1 = 0;
```

```
    loc.reserved2 = 0;
    loc.reserved3 = 0;
    loc.virtualSourceCount = 0;

    err = PtrToHand(&loc, &h, sizeof(loc));
    err = SetTrackSoundLocalizationSettings(t, h);

    DisposeHandle(h);
}
```

GetTrackSoundLocalizationSettings

`GetTrackSoundLocalizationSettings` returns a handle containing a copy of the current 3D Sound settings for the specified track.

```
pascal OSErr GetTrackSoundLocalizationSettings (Track theTrack, Handle
                    *settings)
```

`theTrack` Identifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack`.

`settings` The settings you want to retrieve, in the format of a Sound Sprockets `SSpLocalizationData` record.

DISCUSSION

If there are no 3D Sound settings, the returned handle is set to `nil`. The caller of this routine is reponsible for disposing of the returned handle.

Functions for Editing Movies

PasteHandleIntoMovie

As of QuickTime 1.6.1, the `PasteHandleIntoMovie` function supports a user settings dialog box for import operations. Your application controls whether this dialog appears by setting the value of the `flags` parameter in the `PasteHandleIntoMovie` function. This function supports the following new flag:

`showUserSettingsDialog`
: Controls whether the user settings dialog for the specified import operation can appear. Set this flag to 1 to display the user settings dialog.

Adding Samples to Media Structures

SetMediaDefaultDataRefIndex

The `SetMediaDefaultDataRefIndex` function allows you to specify which of a media's data references is to be accessed during an editing session.

```
pascal OSErr SetMediaDefaultDataRefIndex (Media theMedia, short index);
```

`theMedia`
: Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia`.

`index`
: Specifies the data reference to access. Values of the `index` parameter range from 1 to the number of data references in the media (you can determine the number of data references by calling the `GetMediaDataRefCount` function). Once set, the default data reference index persists. Set this parameter to 0 to revert to the media's default data reference.

DESCRIPTION

Prior to QuickTime 2.0, the Movie Toolbox did not allow the creation of tracks that have data in several files. Therefore, there was not a mechanism for controlling which data reference is affected by a media editing session. The `SetMediaDefaultDataRefIndex` function allows you to specify the index of the data reference to be edited. After calling this function, you can start editing that data reference by calling the `BeginMediaEdits` function.

RESULT CODES

`invalidMedia`	–2008	The media is corrupted or invalid
`badDataRefIndex`	–2050	Data reference index value is invalid

SetMediaPreferredChunkSize

The `SetMediaPreferredChunkSize` function allows you to specify a maximum chunk size for a media.

```
pascal OSErr SetMediaPreferredChunkSize (Media theMedia, long
                    maxChunkSize);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia`.

`maxChunkSize` Specifies the maximum chunk size, in bytes.

DISCUSSION

The term *chunk* refers to the collection of sample data that is added to a movie when you call the `AddMediaSample` function. When QuickTime loads a movie for playback, it loads the data a chunk at a time. Consequently, both the size and number of chunks in a movie can affect playback performance. The Movie Toolbox tries to optimize playback performance by consolidating adjacent sample references into a single chunk (up to the limit you prescribe with this function).

RESULT CODES

noMediaHandler	–2006	Media has no media handler
invalidMedia	–2008	The media is corrupted or invalid

GetMediaPreferredChunkSize

The GetMediaPreferredChunkSize function allows you to retrieve the maximum chunk size for a media.

```
pascal OSErr GetMediaPreferredChunkSize (Media theMedia, long
                    *maxChunkSize);
```

theMedia — Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as NewTrackMedia and GetTrackMedia.

maxChunkSize — Specifies a field to receive the maximum chunk size, in bytes.

RESULT CODES

noMediaHandler	–2006	Media has no media handler
invalidMedia	–2008	The media is corrupted or invalid

Editing Tracks

The Movie Toolbox contains one new function for editing tracks.

AddEmptyTrackToMovie

The AddEmptyTrackToMovie function duplicates a track from a movie into the same movie, or into another movie. The newly created track has the same media type and track settings as the specified track. However, no data is copied from the source track to the new track.

To copy data from the source track to the new track, use the `InsertTrackSegment` function after calling `AddEmptyTrackToMovie`.

```
pascal OSErr AddEmptyTrackToMovie(Track srcTrack,
                    Movie dstMovie,
                    Handle dataRef,
                    OSType dataRefType,
                    Track *dstTrack);
```

`srcTrack` Specifies the source track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack`.

`dstMovie` Specifies the destination movie for this operation. This can be the same movie as the source track or a different movie.

`dataRef` Contains a handle to the data reference. The type of information stored in the handle depends upon the data reference type specified by the `dataRefType` parameter.

`dataRefType` Specifies the type of data reference. If the data reference is an alias, you must set the parameter to `rAliasType` , indicating that the reference is an alias.

`dstTrack` The newly created track's identifier is returned in this parameter. If `AddEmptyTrackToMovie` fails, the resulting track identifier is set to `nil`.

DISCUSSION

The `AddEmptyTrackToMovie` function returns the newly created, empty track. This function has been available since QuickTime 2.0.

Using the Full Screen

QuickTime 2.1 introduced two functions that you can use to put a device into full-screen mode (that is, select where and when the menu bar is not visible).

BeginFullScreen

You can use the `BeginFullScreen` function to begin full-screen mode for a specified monitor.

```
pascal OSErr BeginFullScreen (
                    Ptr *restoreState,
                    GDHandle whichGD,
                    short *desiredWidth,
                    short *desiredHeight,
                    WindowPtr *newWindow,
                    RGBColor *eraseColor,
                    long flags);
```

`restoreState` On exit, a pointer to a block of private state information that contains information on how to return from full-screen mode. This value is passed to `EndFullScreen` to enable it to return the monitor to its previous state.

`whichGD` A handle to the graphics device to put into full-screen mode. Set this parameter to `nil` to select the main screen.

`desiredWidth` On entry, a pointer to a short integer that contains the desired width, in pixels, of the images to be displayed. On exit, that short integer is set to the actual number of pixels that can be displayed horizontally. Set this parameter to 0 to leave the width of the display unchanged.

`desiredHeight` On entry, a pointer to a short integer that contains the desired height, in pixels, of the images to be displayed. On exit, that short integer is set to the actual number of pixels that can be displayed vertically. Set this parameter to 0 to leave the height of the display unchanged.

`newWindow` On entry, a window-creation value. If this parameter is `nil`, no window is created for you. If this parameter has any other value, `BeginFullScreen` creates a new window that is large enough to fill the entire screen and returns a pointer to that window in this parameter. You should not dispose of that window yourself; instead, `EndFullScreen` will do so.

eraseColor — The color to use when erasing the full-screen window created by BeginFullScreen if newWindow is not nil on entry. If this parameter is nil, BeginFullScreen uses black when initially erasing the window's content area.

flags — The flags parameter specifies a set of bit flags that control certain aspects of the full-screen mode. QuickTime defines these constants that you can use in the flags parameter.

```
enum {
    fullScreenHideCursor            = 1L << 0,
    fullScreenAllowEvents           = 1L << 1,
    fullScreenDontChangeMenuBar     = 1L << 2,
    fullScreenPreflightSize         = 1L << 3
};
```

Flag description

fullScreenHideCursor

If this flag is set, BeginFullScreen hides the cursor. This is useful if you are going to play a QuickTime movie and do not want the cursor to be visible over the movie.

fullScreenAllowEvents

If this flag is set, your application intends to allow other applications to run (by calling WaitNextEvent to grant them processing time). In this case, BeginFullScreen does not change the monitor resolution, because other applications might depend on the current resolution.

fullScreenDontChangeMenuBar

If this flag is set, BeginFullScreen does not hide the menu bar. This is useful if you want to change the resolution of the monitor but still need to allow the user to access the menu bar.

fullScreenPreflightSize

If this flag is set, BeginFullScreen doesn't change any monitor settings, but returns the actual height and width that it would use if this bit were not set. This allows applications to test for the availability of a monitor setting without having to switch to it.

DISCUSSION

The `BeginFullScreen` function returns, in the `restoreState` parameter, a pointer to a block of private state information that indicates how to return from full-screen mode. You pass that pointer as a parameter to the `EndFullScreen` function (described next).

The Macintosh Interface Standard states that the menu bar must always be present, and that information must always appear in windows. However, many multimedia applications have chosen to change the look and feel of the interface based on there needs. The number of details to keep track of when doing this continues to increase. To help solve this problem, QuickTime 2.1 added functions to put a GDevice into full screen mode.

EndFullScreen

You can use the `EndFullScreen` function to end full-screen mode for a graphics device.

```
pascal OSErr EndFullScreen (Ptr fullState, long flags);
```

`fullState`	The pointer to private state information returned by a previous call to `BeginFullScreen`.
`flags`	Reserved. Set this parameter to `nil`.

DISCUSSION

The `EndFullScreen` function restores the graphics device and other settings to the state specified by the private state information pointed to by the `fullState` parameter. The resulting state is that that was in effect prior to the immediately previous call to the `BeginFullScreen` function.

Handling Update Events

QuickTime 2.1 introduced a new function, `InvalidateMovieRegion`, to use in place of the `UpdateMovie` function to indicate the area of a movie that needs to be redrawn.

InvalidateMovieRegion

Use the new `InvalidateMovieRegion` function instead of the `UpdateMovie` function to invalidate a small area of a movie. `InvalidateMovieRegion` marks all areas of the movie that intersect the `invalidRgn` parameter. The next time you call the `MoviesTask` function, the Movie Toolbox redraws the marked areas.

```
pascal OSErr InvalidateMovieRegion (
                    Movie theMovie,
                    RgnHandle invalidRgn);
```

`theMovie` Identifies the movie whose area you wish to invalidate. Your application obtains this movie identifier from such functions as NewMovie, NewMovieFromFile, and NewMovieFromHandle.

`invalidRgn` Contains a region indicating the area of the movie to invalidate. If necessary, QuickTime will make a copy of this region. To invalidate the entire movie area, pass `nil` for this parameter.

DESCRIPTION

The `InvalidateMovieRegion` function provides a way to invalidate a portion of the movie's area instead of its entire area, as does `UpdateMovie`. This allows for higher performance update handling when a movie has many tracks, or covers a large area. For handling of update events, applications should continue to use `UpdateMovie`.

RESULT CODES

`invalidMovie`	–2010	The movie is corrupted or invalid

Handling Media Sample References

You could always use `GetMediaSampleReference` to access samples in a movie one at a time. QuickTime 2.1 introduced `GetMediaSampleReferences` (note that this is the plural form of the `GetMediaSampleReference` function), that you can use to obtain information about groups of samples. QuickTime 2.1 also introduced `AddMediaSampleReferences` that you can use to work with groups of samples that have already been added to a movie.

GetMediaSampleReferences

The `GetMediaSampleReferences` function allows your application to obtain reference information about groups of samples that are stored in a movie.

```
pascal OSErr GetMediaSampleReferences (
                    Media theMedia,
                    TimeValue time,
                    TimeValue *sampleTime,
                    SampleDescriptionHandle sampleDescriptionH,
                    long *sampleDescriptionIndex,
                    long maxNumberOfEntries,
                    long *actualNumberofEntries,
                    SampleReferencePtr sampleRefs);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia`. For information about these functions, see *Inside Macintosh: QuickTime.*

`time` Specifies the starting time of the sample references to be retrieved. You must specify this value in the media's time scale.

`sampleTime` Contains a pointer to a time value. The `GetMediaSampleReferences` function updates this time value to indicate the actual time of the first returned sample data. If you are not interested in this information, set this parameter to `nil`.

`sampleDescriptionH`
Contains a handle to a sample description. The `GetMediaSampleReference` function returns the sample description corresponding to the returned sample data. The function resizes this handle as appropriate. If you do not want the sample description, set this parameter to `nil`.

`GetMediaSampleReferences` only returns a single sample description. If the sample description changes within the media, `GetMediaSampleReferences` will return only as many samples as use a single sample description. You must call it again to get the next group of samples using the next sample description.

`sampleDescriptionIndex`
Contains a pointer to a long integer. The `GetMediaSampleReferences` function returns an index value to the sample descriptions that correspond to the returned sample data. You can use this index to retrieve the media sample descriptions with the `GetMediaSampleDescription` function. If you do not want this information, set this parameter to `nil`.

`maxNumberOfEntries`
Specifies the maximum number of entries to be returned. The sample references pointer provided by the `sampleRefs` parameter must be large enough to receive the number of entries specified by this parameter. The Movie Toolbox does not return more entries than you specify with this parameter. It may, however, return fewer.

`actualNumberOfEntries`
Contains a pointer to a long integer. The `GetMediaSampleReferences` function updates the field referred to by this parameter with the number of entries referred to by the returned reference.

`sampleRefs` Contains a pointer to the number of `SampleReferenceRecords` specified in the `maxNumberOfEntries` parameter. On return from this call, the number of sample reference records indicated by the value returned in `actualNumberOfEntries` will be filled in.

DESCRIPTION

Using this function instead of `GetMediaSampleReference` can greatly increase the performance of operations which need access to information about each sample in a movie. No information is returned from this call that wasn't previously available from `GetMediaSampleReference`.

RESULT CODES

`invalidMedia`	-2008	This media is corrupted or invalid.

Memory Manager errors

AddMediaSampleReferences

The `AddMediaSampleReferences` function allows your application to add groups of samples to a movie data file.

```
pascal OSErr AddMediaSampleReferences (
                    Media theMedia,
                    SampleDescriptionHandle sampleDescriptionH,
                    long numberOfSamples,
                    SampleReferencePtr sampleRefs,
                    TimeValue *sampleTime);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia`. For information about these functions, see *Inside Macintosh: QuickTime.*

`sampleDescriptionH`
Contains a handle to a sample description. Some media structures may require sample descriptions. There are different sample descriptions for different types of samples. For example, a media that contains compressed video requires that you supply an image description. A media that contains sound requires that you supply a sound description structure. For information about the Image Compression Manager and the sound description structure, see *Inside Macintosh: QuickTime.*

If you do not want the sample description, set this parameter to `nil`.

`numberOfSamples`
Specifies the number of samples contained in the reference. For details, see the `AddMediaSample` function description in *Inside Macintosh: QuickTime*.

`sampleRefs` Contains a pointer to the number of `SampleReferenceRecords` specified in the `numberOfSamples` parameter.

`sampleTime` Contains a pointer to a time value. After adding the reference to the media, the `AddMediaSampleReferences` function returns the time where the reference was inserted in the time value referred to by this parameter. If you do not want to receive this information, set this parameter to `nil`.

DISCUSSION

Using this function instead of `AddMediaSampleReference` can greatly improve the performance of operations which involve adding a large number of samples to a movie at one time. `AddMediaSampleReferences` provides no capabilities that weren't previously available with `AddMediaSampleReference`.

RESULT CODES

`invalidMedia`	-2008	This media is corrupted or invalid.

Memory Manager errors

Managing the Video Frame Playback Rate

QuickTime 2.1 introduced two new functions for determining the rate at which a QuickTime movie plays back each video frame. You should use these functions for debugging.

GetVideoMediaStatistics

The `GetVideoMediaStatistics` function returns the play-back frame rate of a movie. This call can only be used on video or MPEG media handlers.

```
pascal Fixed GetVideoMediaStatistics (
                    MediaHandler mh);
```

`mh` Contains a reference to a video media handler. You obtain this reference from the `GetMediaHandler` function.

function result Returns the movie's play-back frame rate in frames-per-second.

DESCRIPTION

The `GetVideoMediaStatistics` returns the average frame rate since the last time `ResetVideoMediaStatistics` was called. Because of sampling errors, the values returned from `GetVideoMediaStatistics` are accurate only after waiting at least one second after calling `ResetVideoMediaStatistics`.

Note
Because not all QuickTime movies have a constant frame rate, the results of this call can be difficult to correctly interpret. For this reason, the results of this function should not be displayed in a place where a novice user is likely to see it.

ResetVideoMediaStatistics

Use the `ResetVideoMediaStatistics` function to reset the video media handler's counters before using `GetVideoMediaStatistics` to determine the frame rate of a movie. This call can only be used on video or MPEG media handlers.

```
pascal HandlerError ResetVideoMediaStatistics (
                    MediaHandler mh);
```

`mh` Contains a reference to a video media handler. You obtain this reference from the `GetMediaHandler` function.

function result Returns a handle error.

DESCRIPTION

The `ResetVideoMediaStatistics` function resets the video media handler's frame rate counters.

RESULT CODES

`badComponentInstance`	0x80008001	Invalid component instance specified.

Manipulating Media Input Maps

The Movie Toolbox contains two functions for maintaining media input maps: `GetMediaInputMap` and `SetMediaInputMap`.

Each track has particular attributes such as size, position, and volume associated with it. The media input map of that track describes where the variable parameters are stored so that modifier tracks know where to send their

data. When a track is copied, its input map is also copied. `CopyTrackSettings` also transfers the media input map.

GetMediaInputMap

The `GetMediaInputMap` function returns a copy of the input map associated with the specified media. The caller is responsible for disposing of the input map with `QTDisposeAtomContainer`.

```
pascal OSErr GetMediaInputMap (
                    Media theMedia,
                    QTAtomContainer *inputMap,);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia`.

`inputMap` Specifies the media input map for this operation. You must dispose of the map referred to by this parameter when you are done with it using `QTDisposeAtomContainer`.

DISCUSSION

Use the `GetMediaInputMap` function to specify the media you want to get so you can modify its input map.

RESULT CODES

invalidMedia	–2008	The media is corrupted or invalid
paramErr	–50	Invalid parameter specified
memFullErr	-108	Not enough room in heap zone

SetMediaInputMap

The SetMediaInputMap function replaces the media's existing input map with the given input map.

```
pascal OSErr SetMediaInputMap (
                    Media theMedia,
                    QTAtomContainer inputMap);
```

theMedia — Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as NewTrackMedia and GetTrackMedia.

inputMap — Specifies the media input map for this operation. If the input map is set to nil, the media's input map is reset to an empty input map.

DISCUSSION

Use the SetMediaInputMap function to specify the media you want to set so you can modify or empty its input map.

SetMediaInputMap makes a copy of the inputMap passed to it. Typically, an application will call GetMediaInputMap to get the current input map before modifying it. Use QTNewAtomContainer to create an empty input map. See the description of QTNewAtomContainer later in this chapter.

RESULT CODES

invalidMedia	–2008	The media is corrupted or invalid
paramErr	–50	Invalid parameter specified
memFullErr	-108	Not enough room in heap zone

Media Functions

Selecting Data Handlers

GetDataHandler

The GetDataHandler function allows you to retrieve the best data handler component to use with a given data reference.

```
pascal Component GetDataHandler (Handle dataRef,
                    OSType dataHandlerSubType, long flags);
```

dataRef — Contains a handle to the data reference. The type of information stored in the handle depends upon the data reference type specified by the dataHandlerSubType parameter.

dataHandlerSubType — Identifies both the type of data reference and, by implication, the component subtype value assigned to the data handler components that operate on data references of that type.

flags — Indicates the way in which you intend to use the data handler component. Note that not all data handlers necessarily support all services—for example, some data handler components may not support streaming writes.

The following flags are defined (set the appropriate flags to 1):

kDataHCanRead — Specifies that you intend to use the data handler component to read data.

kDataHCanWrite
: Specifies that you intend to use the data handler component to write data.

kDataHCanStreamingWrite
: Indicates that you intend to do streaming writes (as part of a movie-capture operation, for example).

DESCRIPTION

Once you have used this function to get information about the best data handler component for your data reference, you can open and use the component using Component Manager functions. See "Data Handler Components," for more information.

If the function returns a value of `nil`, the Movie Toolbox was unable to find an appropriate data handler component. For more information about the error, call the `GetMoviesError` Movie Toolbox function.

RESULT CODES

Memory Manager errors

Timecode Media Handler Functions

This section discusses the functions and structures that allow you to use the timecode media handler.

The timecode media handler allows QuickTime movies to store timing information that is derived from the movie's original source material. Every QuickTime movie contains QuickTime-specific timing information, such as frame duration. This information affects how QuickTime interprets and plays the movie.

The timecode media handler allows QuickTime movies to store additional timing information that is not created by or for QuickTime. This additional timing information would typically be derived from the original source material, say as a SMPTE timecode. In essence, you can think of the timecode media handler as providing a link between the "digital" QuickTime-specific timing information and the original "analog" timing information from the source material.

A movie's timecode is stored in a timecode track. Timecode tracks contain:

- Source identification information (this identifies the source, say, a given videotape)
- Timecode format information (this specifies the characteristics of the timecode and how to interpret the timecode information)
- Frame numbers (these allow QuickTime to map from a given movie time—in terms of QuickTime time values—to its corresponding timecode value)

Apple has defined the information that is stored in the track in a manner that is independent of any specific timecode standard. The format of this information is sufficiently flexible to accommodate all known timecode standards, including, for example, SMPTE timecode. The timecode format information provides QuickTime the parameters for understanding the timecode and converting QuickTime time values into timecode time values (and vice versa).

One key timecode attribute relates to the technique used to synchronize timecode values with video frames. Most video source material is recorded at whole-number frame rates. For example, both PAL and SECAM video contains exactly 25 frames per second. However, some video source material is not recorded at whole-number frame rates. In particular, NTSC color video contains 29.97 frames per second (though it is typically referred to as 30 frames-per-second video). However, NTSC timecode values correspond to the full 30 frames-per-second rate (this is a holdover from NTSC black-and-white video). For such video sources, you need a mechanism that corrects the skew that will develop over time between timecode values and actual video frames.

A common method for maintaining synchronization between timecode values and video data is called dropframe. Contrary to its name, the dropframe technique actually skips timecode values at a predetermined rate in order to keep the timecode and video data synchronized. It does not actually drop video frames. In NTSC color video, which uses the dropframe technique, the timecode values skip two frame values every minute, except for minute values that are evenly divisible by ten. So NTSC timecode values, which are expressed as HH:MM:SS:FF (hours, minutes, seconds, frames) skip from 00:00:59:29 to 00:01:00:02 (skipping 00:01:00:00 and 00:01:00:01). There is a flag in the timecode definition structure that indicates whether the timecode uses the dropframe technique.

You can have the Movie Toolbox display the timecode when a movie is played. Use the `TCSetTimeCodeFlags` function to turn the timecode display on and off. Note that the timecode track must be enabled for this display to work.

You store the timecode's source identification information in a user data item. Create a user data item with a type value of `TCSourceRefNameType`. Store the source information as a text string. This information might contain the name of the videotape from which the movie was created, for example. For more information about working with user data, see *Inside Macintosh: QuickTime*.

The timecode media handler provides functions that allow you to manipulate the source identification information. The following sample code demonstrates one way to set the source tape name in a timecode media's sample description.

```
void setTimeCodeSourceName (Media timeCodeMedia,
                            TimeCodeDescriptionHandle tcdH,
                            Str255 tapeName, ScriptCode
tapeNameScript)

{
    UserData srcRef;

    if (NewUserData(&srcRef) == noErr) {
        Handle nameHandle;

        if (PtrToHand(&tapeName[1], &nameHandle, tapeName[0]) == noErr) {
            if (AddUserDataText (srcRef, nameHandle,'name', 1,
                                 tapeNameScript) == noErr) {
                TCSetSourceRef (GetMediaHandler (timeCodeMedia),
                                 tcdH,
                                 srcRef);
            }
            DisposeHandle(nameHandle);
        }
        DisposeUserData(srcRef);
    }
}
```

You create a timecode track and media in the same manner that you create any other track. Call the `NewMovieTrack` function to create the timecode track, and use the `NewTrackMedia` function to create the track's media. Be sure to specify a media type value of `TimeCodeMediaType` when you call the `NewTrackMedia` function.

You define the relationship between a timecode track and one or more movie tracks using the Movie Toolbox's new track reference functions (see "Track

References" and "Functions for Working With Track References" elsewhere in this chapter for more information). You then proceed to add samples to the track, as appropriate.

Each sample in the timecode track provides timecode information for a span of movie time. The sample includes duration information. As a result, you typically add each timecode sample after you have created the corresponding content track or tracks.

The timecode media sample description contains the control information that allows QuickTime to interpret the samples. This includes the timecode format information. The actual sample data contains a frame number that identifies one or more content frames that use this timecode. Stored as a long, this value identifies the first frame in the group of frames that use this timecode. In the case of a movie made from source material that contains no edits, you would only need one sample. When the source material contains edits, you typically need one sample for each edit, so that QuickTime can re-sync the timecode information with the movie. Those samples contain the frame numbers of the frames that begin each new group of frames.

The timecode description structure defines the format and content of a timecode media sample description.

```
typedef struct TimeCodeDescription {
    long                descSize;   /* size of the structure */
    long                dataFormat; /* sample type */
    long                resvd1;         /* reserved--set to 0 */
    short               resvd2;         /* reserved--set to 0 */
    short               dataRefIndex;/* data reference index */
    long                flags;          /* reserved--set to 0 */
    TimeCodeDef timeCodeDef;/* timecode format information */
    long                srcRef[1];  /* source information */
} TimeCodeDescription, *TimeCodeDescriptionPtr,
**TimeCodeDescriptionHandle;
```

Field descriptions

`descSize`	Specifies the size of the sample description, in bytes.
`dataFormat`	Indicates the sample description type (`TimeCodeMediaType`).
`resvd1`	Reserved for use by Apple. Set this field to 0.
`resvd2`	Reserved for use by Apple. Set this field to 0.

dataRefIndex	Contains an index value indicating which of the media's data references contains the sample data for this sample description.
flags	Reserved for use by Apple. Set this field to 0.
timeCodeDef	Contains a timecode definition structure that defines timecode format information.
srcRef	Contains the timecode's source information. This is formatted as a user data item that is stored in the sample description. The media handler provides functions that allow you to get and set this data.

The timecode definition structure contains the timecode format information. This structure is defined as follows:

```
typedef struct TimeCodeDef {
    long                flags;          /* timecode control flags */
    TimeScale   fTimeScale; /* timecode's time scale */
    TimeValue   frameDuration;/* how long each frame lasts */
    unsigned charnumFrames;/* number of frames per second */
} TimeCodeDef;
```

Field descriptions

flags	Contains flags that provide some timecode format information. The following flags are defined:
tcDropFrame	Indicates that the timecode "drops" frames occasionally in order to stay in sync. Some timecodes run at other than a whole number of frames per second. For example, NTSC video runs at 29.97 frames per second. In order to resynchronize between the timecode rate and a 30 frames-per-second playback rate, the timecode will drop a frame at a predictable time (in much the same way that leap years keep the calendar in sync). Set this flag to 1 if the timecode uses the dropframe technique.
tc24HourMax	Indicates that the timecode values wrap at 24 hours. Set this flag to 1 if the timecode hour value wraps (that is, returns to 0) at 24 hours.
tcNegTimesOK	Indicates that the timecode supports negative time values. Set this flag to 1 if the timecode allows negative values.
tcCounter	Indicates that the timecode should be interpreted as a simple counter, rather than as a time value. This allows the

	timecode to contain either time information or counter (such as a tape counter) information. Set this flag to 1 if the timecode contains counter information.
`fTimeScale`	Contains the time scale for interpreting the `frameDuration` field. This field indicates the number of time units per second.
`frameDuration`	Specifies how long each frame lasts, in the units defined by the `fTimeScale` field.
`numFrames`	Indicates the number of frames stored per second. In the case of timecodes that are interpreted as counters, this field indicates the number of frames stored per timer "tick."

The best way to understand how to format and interpret the timecode definition structure is to consider an example. If you were creating a movie from an NTSC video source recorded at 29.97 frames per second, using SMPTE timecode, you would format the timecode definition structure as follows:

```
    TimeCodeDef.flags = tcDropFrame | tc24HourMax;
    TimeCodeDef.fTimeScale = 2997;/* units */
    TimeCodeDef.frameDuration = 100;/* relates units to frames */
    TimeCodeDef.numFrames = 30;/* whole frames per second */ i).timecode
media handler:timecode definition structure;
```

The movie's natural frame rate of 29.97 frames per second is obtained by dividing the `fTimeScale` value by the `frameDuration` (2997÷100). Note that the `flags` field indicates that the timecode uses the dropframe technique to resync the movie's natural frame rate of 29.97 frames per second with its playback rate of 30 frames per second.

Given a timecode definition, you can freely convert from frame numbers to time values and from time values to frame numbers. For a time value of 00:00:12:15 (HH:MM:SS:FF), you would obtain a frame number of 375 (12*30 + 15). The timecode media handler provides a number of functions that allow you to perform these conversions.

When you use the timecode media handler to work with time values, the media handler uses timecode records to store the time values. The timecode record allows you to interpret the time information as either a time value (HH:MM:SS:FF) or a counter value. The timecode record is defined as follows:

```
typedef union TimeCodeRecord {
    TimeCodeTimet;          /* value interpreted as time */
    TimeCodeCounterc;       /* value interpreted as counter */
} TimeCodeRecord;

typedef struct TimeCodeTime {
    unsigned charhours;     /* time: hours */
    unsigned charminutes;   /* time: minutes */
    unsigned charseconds;   /* time: seconds */
    unsigned charframes;    /* time: frames */
} TimeCodeTime;

typedef struct TimeCodeCounter {
    long                counter;        /* counter value */
} TimeCodeCounter;
```

Note that, when you are working with timecodes that allow negative time values, the `minutes` field of the `TimeCodeTime` structure (`TimeCodeRecord.t.minutes`) indicates whether the time value is positive or negative. If the `tctNegFlag` bit of the `minutes` field is set to 1, the time value is negative.

TCGetCurrentTimeCode

The `TCGetCurrentTimeCode` function retrieves the timecode and source identification information for the current movie time.

```
pascal HandlerError TCGetCurrentTimeCode (MediaHandler mh, long
                    *frameNum, TimeCodeDef *tcdef, TimeCodeRecord
                    *tcrec, UserData *srcRefH);
```

`mh` Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.

`frameNum` Contains a pointer to a field that is to receive the current frame number. Set this field to `nil` if you do not want to retrieve the frame number.

tcdef	Contains a pointer to a timecode definition structure. The media handler returns the movie's timecode definition information. Set this parameter to nil if you do not want this information.
tcrec	Contains a pointer to a timecode record structure. The media handler returns the current time value. Set this parameter to nil if you do not want this information.
srcRefH	Contains a pointer to a field that is to receive a handle containing the source information. It is your responsibility to dispose of this user data when you are done with it. Set this field to nil if you do not want this information.

RESULT CODES

invalidTime	–2015	This time value is invalid

TCGetTimeCodeAtTime

The TCGetTimeCodeAtTime function returns a track's timecode information corresponding to a specific media time.

```
pascal HandlerError TCGetTimeCodeAtTime (MediaHandler mh, TimeValue
                    mediaTime, long *frameNum, TimeCodeDef *tcdef,
                    TimeCodeRecord *tcdata, UserData *srcRefH);
```

mh	Specifies the timecode media handler. You obtain this identifier by calling the GetMediaHandler function.
mediaTime	Specifies the time value for which you want to retrieve timecode information. This time value is expressed in the media's time coordinate system.
frameNum	Contains a pointer to a field that is to receive the current frame number. Set this field to nil if you do not want to retrieve the frame number.

tcdef	Contains a pointer to a timecode definition structure. The media handler returns the movie's timecode definition information. Set this parameter to nil if you do not want this information.
tcrec	Contains a pointer to a timecode record structure. The media handler returns the current time value. Set this parameter to nil if you do not want this information.
srcRefH	Contains a pointer to a field that is to receive a handle containing the source information. It is your responsibility to dispose of this user data when you are done with it. Set this field to nil if you do not want this information.

RESULT CODES

invalidTime	–2015	This time value is invalid

Memory Manager errors

TCTimeCodeToFrameNumber

The TCTimeCodeToFrameNumber function converts a timecode time value into its corresponding frame number.

```
pascal HandlerError TCTimeCodeToFrameNumber (MediaHandler mh,
                    TimeCodeDef *tcdef, TimeCodeRecord *tcrec,
                    long *frameNumber);
```

mh	Specifies the timecode media handler. You obtain this identifier by calling the GetMediaHandler function.
tcdef	Contains a pointer to the timecode definition structure to use for the conversion.
tcrec	Contains a pointer to the timecode record structure containing the time value to convert.
frameNumber	Contains a pointer to a field that is to receive the frame number that corresponds to the time value in the tcrec parameter.

RESULT CODES

paramErr	–50	Invalid parameter specified

TCFrameNumberToTimeCode

The TCFrameNumberToTimeCode function converts a frame number into its corresponding timecode time value.

```
pascal HandlerError TCFrameNumberToTimeCode (MediaHandler mh, long
                    frameNumber, TimeCodeDef *tcdef, TimeCodeRecord
                    *tcrec);
```

mh	Specifies the timecode media handler. You obtain this identifier by calling the GetMediaHandler function.
frameNumber	Specifies the frame number that is to be converted.
tcdef	Contains a pointer to the timecode definition structure to use for the conversion.
tcrec	Contains a pointer to the timecode record structure that is to receive the time value.

RESULT CODES

paramErr	–50	Invalid parameter specified

TCTimeCodeToString

The TCTimeCodeToString function converts a time value into a text string (HH:MM:SS:FF). If the timecode uses the dropframe technique, the separators are semi-colons (;) rather than colons (:).

```
pascal HandlerError TCTimeCodeToString(MediaHandler mh, TimeCodeDef
                    *tcdef, TimeCodeRecord *tcrec, StringPtr tcStr);
```

mh	Specifies the timecode media handler. You obtain this identifier by calling the GetMediaHandler function.
tcdef	Contains a pointer to the timecode definition structure to use for the conversion.
tcrec	Contains a pointer to the timecode record structure to use for the conversion.
tcStr	A pointer to a text string that is to receive the converted time value.

RESULT CODES

paramErr	–50	Invalid parameter specified

TCSetSourceRef

The TCSetSourceRef function allows you to change the source information in the timecode media sample reference.

```
pascal HandlerError TCSetSourceRef (MediaHandler mh,
                    TimeCodeDescriptionHandle tcdH, UserData srefH);
```

mh	Specifies the timecode media handler. You obtain this identifier by calling the GetMediaHandler function.
tcdH	Specifies a handle containing the timecode media sample reference that is to be updated.
srefH	Specifies a handle to the source information to be placed in the sample reference. It is your application's responsibility to dispose of this user data when you are done with it.

RESULT CODES

paramErr	–50	Invalid parameter specified

Memory Manager errors

TCGetSourceRef

The `TCGetSourceRef` function allows you to retrieve the source information from the timecode media sample reference.

```
pascal HandlerError TCGetSourceRef (MediaHandler mh,
                    TimeCodeDescriptionHandle tcdH, UserData *srefH);
```

`mh` Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.

`tcdH` Specifies a handle containing the timecode media sample reference for this operation.

`srefH` Specifies a pointer to a handle that will receive the source information. It is your application's responsibility to dispose of this user data when you are done with it.

RESULT CODES

`paramErr`	–50	Invalid parameter specified

Memory Manager errors

TCSetTimeCodeFlags

The `TCSetTimeCodeFlags` function allows you to change the flags that affect how the Movie Toolbox handles the timecode information.

```
pascal HandlerError TCSetTimeCodeFlags (MediaHandler mh, long flags,
                    long flagsMask);
```

`mh` Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.

`flags` Specifies the new flag values. The following flags are defined:

tcdfShowTimeCode

Controls the display of timecode information. Set this flag to 1 to cause timecode information to be displayed when the movie plays. Set this flag to 0 to turn off the display.

Note that the timecode track must be enabled in order for the timecode information to be displayed.

flagsMask — Specifies which of the flag values are to change. The media handler modifies only those flag values that correspond to bits that are set to 1 in this parameter. Use the flag values from the flags parameter. For example, in order to turn off timecode display, you would set the tcdfShowTimeCode flag to 1 in the flagsMask parameter, and to 0 in the flags parameter.

TCGetTimeCodeFlags

The TCGetTimeCodeFlags function allows you to retrieve the timecode control flags.

```
pascal HandlerError TCGetTimeCodeFlags (MediaHandler mh, long *flags;
```

mh — Specifies the timecode media handler. You obtain this identifier by calling the GetMediaHandler function.

flags — Contains a pointer to a field that is to receive the control flags. The following flags are defined:

tcdfShowTimeCode

Controls the display of timecode information. If this flag is set to 1, the timecode information is displayed when the movie is played.

Note that the timecode track must be enabled in order for the timecode information to be displayed.

TCSetDisplayOptions

The `TCSetDisplayOptions` function allows you to set the text characteristics that apply to timecode information that is displayed in a movie.

```
pascal HandlerError TCSetDisplayOptions (MediaHandler mh,
                    TCTextOptionsPtr textOptions);
```

`mh` Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.

`textOptions` Contains a pointer to a text options structure. This structure contains font and style information.

DESCRIPTION

You provide the text style information in a text options structure. This structure is defined as follows (for more information about working with text characteristics, see *Inside Macintosh: Text*):

```
typedef struct TCTextOptions {
    short               txFont;         /* font */
    short               txFace;         /* font style */
    short               txSize;         /* font size */
    RGBColor        foreColor;  /* foreground color */
    RGBColor        backColor;  /* background color */
} TCTextOptions, *TCTextOptionsPtr;
```

Field descriptions

`txFont` Specifies the number of the font.

`txFace` Specifies the font's style (bold, italic, and so on).

`txSize` Specifies the font's size.

`foreColor` Specifies the foreground color.

`backColor` Specifies the background color.

TCGetDisplayOptions

The `TCGetDisplayOptions` function allows you to retrieve the text characteristics that apply to timecode information that is displayed in a movie.

```
pascal HandlerError TCGetDisplayOptions (MediaHandler mh,
                    TCTextOptionsPtr textOptions);
```

`mh` Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.

`textOptions` Contains a pointer to a text options structure. This structure will receive font and style information.

RESULT CODES

`paramErr`	–50	Invalid parameter specified

Media Property Functions

This section discusses functions for setting and retrieving the property atom container of a media handler.

GetMediaPropertyAtom

The `GetMediaPropertyAtom` function retrieves the property atom container of a media handler.

```
pascal OSErr GetMediaPropertyAtom (Media theMedia,
                    QTAtomContainer *propertyAtom)
```

`theMedia` Contains a reference to the media handler for this operation.

`propertyAtom` Contains a pointer to a QT atom container. On return, the atom container contains the property atoms for the track associated with the media handler.

DISCUSSION

You can call the `GetMediaPropertyAtom` to retrieve the properties of the track associated with the specified media handler. The contents of the returned QT atom container are defined by the media handler. The caller is responsible for disposing of the QT atom container.

RESULT CODES

`memFullErr`	-108	Not enough room in heap zone
`invalidMedia`	–2008	The media is corrupted or invalid

SetMediaPropertyAtom

The `SetMediaPropertyAtom` function sets the property atom container of a media handler.

```
pascal OSErr SetMediaPropertyAtom (Media theMedia,
                    QTAtomContainer propertyAtom)
```

`theMedia` Contains a reference to the media handler for this operation.

`propertyAtom` Specifies a QT atom container that contains the property atoms for the track associated with the media handler.

DISCUSSION

You can call the `SetMediaPropertyAtom` to set properties for the track associated with the specified media handler. The contents of the QT atom container are defined by the media handler.

RESULT CODES

memFullErr	–108	Not enough room in heap zone
invalidMedia	–2008	The media is corrupted or invalid

Text Media Handler Functions

QuickTime 1.6.1 added five new flags, two constants, and one new function to the text media handler interface. The new flags and constants are defined in "Text Sample Display Flags," and "Text Sample Types," respectively. The new function is defined in this section.

TextMediaSetTextSampleData

The TextMediaSetTextSampleData function allows you to set values before calling the AddTextSample or AddTESample function.

```
pascal ComponentResult TextMediaSetTextSampleData(
                    MediaHandler mh,
                    void *data,
                    OSType dataType)
```

mh	Contains a reference to the text media handler. You obtain this reference from the GetMediaHandler function.
data	Contains a pointer to the data, defined by the dataType parameter.

RESULT CODES

memFullErr	-108	Not enough room in heap zone
paramErr	-50	Invalid parameter specified

DISCUSSION

The following sample code demonstrates how to use the TextMediaSetTextSampleData function:

```
short trans = 127;
Point dropOffset;
MediaHandler mh;

dropOffset.h = dropOffset.v = 4
TextMediaSetTextSampleData(mh,(void *)&dropOffset,dropShadowOffsetType);
TextMediaSetTextSampleData(mh,(void *)&trans,dropShadowTranslucencyType);
```

Note
Be sure to turn on the `dfDropShadow` display flag after you call `AddTextSample` or `AddTESample`. Passing `nil` for the `textColor` parameter in `AddTextSample` or `AddTESample` defaults to black. Passing `nil` for the `backColor` parameter in `AddTextSample` or `AddTESample` defaults to white.

RESULT CODES

`badComponentInstance`	0x80008001	Invalid component instance specified.

Sprite Toolbox Functions

This section describes the functions provided by the Movie Toolbox for sprite support.

Sprite World Functions

This section describes functions that you use to create and manipulate sprite worlds.

NewSpriteWorld

The `NewSpriteWorld` function creates a new sprite world.

```
pascal OSErr NewSpriteWorld (SpriteWorld *newSpriteWorld,
                    GWorldPtr destination,
                    GWorldPtr spriteLayer,
                    RGBColor *backgroundColor,
                    GWorldPtr background);
```

`newSpriteWorld`
Contains a pointer to a field that is to receive the new sprite world's identifier. On return, this field contains the identifier for the newly created sprite world.

`destination` Contains a pointer to a graphics world to be used as the destination.

`spriteLayer` Contains a pointer to a graphics world to be used as the sprite layer.

`backgroundColor`
Contains a pointer to an RGB color to be used as the background color. If you pass a background graphics world to this function by setting the `background` parameter, you can set this parameter to `nil`.

`background` Contains a pointer to a graphics world to be used as the background. If you pass a background color to this function by setting the `backgroundColor` parameter, you can set this parameter to `nil`.

DISCUSSION

You call this function to create a new sprite world with associated destination and sprite layer graphics worlds, and either a background color or a background graphics world. Once created, you can manipulate the sprite world and add sprites to it using other Sprite Toolbox functions. The sprite world created by this function has an identity matrix. The sprite world does not have a clip shape.

The `newSpriteWorld`, `destination`, and `spriteLayer` parameters are all required. You should specify a background color, a background graphics world, or both.

You should not pass `nil` for both parameters. If you specify both a background graphics world and a background color, the sprite world is filled with the background color before the background sprites are drawn. If no background color is specified, black is the default. If you specify a background graphics world, it should have the same dimensions and depth as the graphics world specified by `spriteLayer`. If you draw to the graphics worlds associated with a sprite world using standard QuickDraw and QuickTime functions, your drawing is erased by the sprite world's background color.

Before calling `NewSpriteWorld`, you should call `LockPixels` on the pixel maps of the sprite layer and background graphics worlds. These graphics worlds must remain valid for the lifetime of the sprite world. The sprite world does not own the graphics worlds that are associated with it; it is the caller's responsibility to dispose of the graphics worlds when they are no longer needed.

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified
`memFullErr`	-108	Not enough memory available
`gWorldsNotSameDepthAndSizeErr`	-2066	Dimensions and pixel depth of the two graphics worlds do not match

DisposeSpriteWorld

The `DisposeSpriteWorld` function disposes of a sprite world.

```
pascal void DisposeSpriteWorld  (SpriteWorld theSpriteWorld);
```

`theSpriteWorld`
Specifies the sprite world to dispose.

DISCUSSION

You call this function to dispose of a sprite world created by the `NewSpriteWorld` function. This function also disposes of all of the sprites associated with the sprite world. This function does not dispose of the graphics worlds associated with the sprite world. It is safe to pass nil to this function.

SetSpriteWorldClip

The `SetSpriteWorldClip` function sets a sprite world's clip shape to the specified region.

```
pascal OSErr SetSpriteWorldClip (SpriteWorld theSpriteWorld,
                    RgnHandle clipRgn);
```

`theSpriteWorld`
Specifies the sprite world for this operation.

`clipRgn` Specifies the new clip shape for the sprite world.

DISCUSSION

You call this function to change the clip shape of a sprite world. You may pass a value of `nil` for the `clipRgn` parameter to indicate that there is no longer a clip shape for the sprite world.

The clip shape should be specified in the sprite world's source space, the coordinate system of the sprite layer's graphics world before the sprite world's matrix is applied to it. The specified region is owned by the caller and is not copied by this function.

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified

SetSpriteWorldMatrix

The `SetSpriteWorldMatrix` function sets a sprite world's matrix to the specified matrix.

```
pascal OSErr SetSpriteWorldMatrix (SpriteWorld theSpriteWorld,
                    const MatrixRecord *matrix);
```

`theSpriteWorld`
Specifies the sprite world for this operation.

matrix	Contains a pointer to the new matrix for the sprite world.

DISCUSSION

You call this function to change the matrix of a sprite world. You may pass a value of nil for the matrix parameter to set the sprite world's matrix to an identity matrix.

If a sprite world's matrix is not an identity matrix, translation and scaling may occur when transferring from the sprite layer graphics world to the destination graphics world.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified

SpriteWorldIdle

The SpriteWorldIdle function allows a sprite world to update its invalid areas.

```
pascal OSErr SpriteWorldIdle (SpriteWorld theSpriteWorld,
                    long flagsIn,
                    long *flagsOut);
```

theSpriteWorld	Specifies the sprite world for this operation.
flagsIn	Contains flags describing actions that may take place during the idle.
flagsOut	On return, contains a pointer to flags describing actions that took place during the idle.

DISCUSSION

You call this function to allow a sprite world the opportunity to redraw its invalid areas. This is the only function that causes drawing to occur; you should call it as often as is necessary.

The `flagsIn` parameter contains flags that describe allowable actions during the idle period. For the default behavior, you should set the value of this parameter to 0. The `flagsOut` parameter is optional; if you do not need the information returned by this parameter, set the value of this parameter to `nil`.

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified

InvalidateSpriteWorld

The `InvalidateSpriteWorld` function invalidates a rectangular area of a sprite world.

```
pascal OSErr InvalidateSpriteWorld (SpriteWorld theSpriteWorld,
                    Rect *invalidArea);
```

`theSpriteWorld`
Specifies the sprite world for this operation.

`invalidArea` Contains a pointer to the rectangular area that should be invalidated.

DISCUSSION

Typically, your application calls this function when the sprite world's destination window receives an update event. Invalidating an area of the sprite world will cause the area to be redrawn the next time that `SpriteWorldIdle` is called.

The invalid rectangle pointed to by the `invalidArea` parameter should be specified in the sprite world's source space, the coordinate system of the sprite layer's graphics world before the sprite world's matrix is applied to it. To invalidate the entire sprite world, pass `nil` for this parameter.

When you modify sprite properties, invalidation takes place automatically; you do not need to call `InvalidateSpriteWorld`.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified

SpriteWorldHitTest

The SpriteWorldHitTest function determines whether any sprites are at a specified location in a sprite world.

```
pascal OSErr SpriteWorldHitTest (SpriteWorld theSpriteWorld,
                    long flags,
                    Point loc,
                    Sprite *spriteHit);
```

theSpriteWorld
: Specifies the sprite world for this operation.

flags
: Specifies flags to control the hit testing operation. Allowable flags are spriteHitTestBounds and spriteHitTestImage.

loc
: Specifies a point in the sprite world's display space to test for the existence of a sprite.

spriteHit
: Contains a pointer to a field that is to receive a sprite identifier. On return, this field contains the identifier of the frontmost sprite at the location specified by loc. If no sprite exists at the location, the function sets the value of this parameter to nil.

DISCUSSION

You call this function to determine whether any sprites exist at a specified location in a sprite world's display coordinate system. If you are drawing the sprite world in a window, you should call GlobalToLocal to convert the location to your window's local coordinate system before passing it to SpriteWorldHitTest.

You can pass flags to this function to control the hit testing operation more precisely. For example, you may want the hit test operation to detect a sprite whose bounding box contains the specified location.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified

DisposeAllSprites

The DisposeAllSprites function disposes all sprites associated with a sprite world.

```
pascal void DisposeAllSprites  (SpriteWorld theSpriteWorld);
```

theSpriteWorld
Specifies the sprite world for this operation.

DISCUSSION

This function calls the DisposeSprite function for each sprite associated with the sprite world.

Sprite Functions

This section describes functions that you use to create and manipulate sprites.

NewSprite

The NewSprite function creates a new sprite in the specified sprite world.

```
pascal OSErr NewSprite (Sprite *newSprite,
                    SpriteWorld itsSpriteWorld,
                    ImageDescriptionHandle idh,
                    Ptr imageDataPtr,
                    MatrixRecord *matrix,
                    Boolean visible,
                    short layer);
```

newSprite	Contains a pointer to field that is to receive the new sprite's identifier. On return, this field contains the identifier of the newly created sprite.
itsSpriteWorld	Specifies the sprite world with which the new sprite should be associated.
idh	Contains a handle to an image description of the sprite's image.
imageDataPtr	Contains a pointer to the sprite's image data.
matrix	Contains a pointer to the sprite's matrix. If you pass nil for the matrix parameter, an identity matrix is assigned to the sprite.
visible	Specifies whether the sprite is visible.
layer	Specifies the sprite's layer.

DISCUSSION

You call this function to create a new sprite associated with a sprite world. Once you have created the sprite, you can manipulate it using the SetSpriteProperty function.

The newSprite, itsSpriteWorld, visible, and layer parameters are required. Sprites with lower layer values appear in front of sprites with higher layer values. If you want to create a sprite that is drawn to the background graphics world, you should specify the constant kBackgroundSpriteLayerNum for the layer parameter.

You can defer assigning image data to the sprite by passing nil for both the idh and imageDataPtr parameters. If you choose to defer assigning image data, you must call SetSpriteProperty to assign the image description handle and image data to the sprite before the next call to SpriteWorldIdle. The caller owns the image description handle and the image data pointer; it is the caller's responsibility to dispose of them after it disposes of a sprite.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified
memFullErr	-108	Not enough memory available

DisposeSprite

The DisposeSprite function disposes of a sprite.

```
pascal void DisposeSprite (Sprite theSprite);
```

theSprite The sprite for this operation.

DISCUSSION

You call this function to dispose of a sprite created by the NewSprite function. The image description handle and image data pointer associated with the sprite are not disposed by this function.

InvalidateSprite

The InvalidateSprite function invalidates the portion of a sprite's sprite world that is occupied by the sprite.

```
pascal void InvalidateSprite  (Sprite theSprite);
```

theSprite The sprite for this operation.

DISCUSSION

In most cases, you do not need to call this function. When you call the SetSpriteProperty function to modify a sprite's properties, SetSpriteProperty takes care of invalidating the appropriate regions of the sprite world. However, you might call this function if you change a sprite's image data, but retain the same image data pointer.

SpriteHitTest

The `SpriteHitTest` function determines whether a location in a sprite's display coordinate system intersects the sprite.

```
pascal OSErr SpriteHitTest (Sprite theSprite,
                    long flags,
                    Point loc,
                    Boolean *wasHit);
```

`theSprite`	Specifies the sprite for this operation.
`flags`	Specifies flags to control the hit testing operation. Allowable flags are `spriteHitTestBounds` and `spriteHitTestImage`.
`loc`	Specifies a point in the sprite world's display space to test for the existence of a sprite.
`wasHit`	Contains a pointer to a Boolean. On return, the value of the Boolean is `true` if the sprite is at the specified location.

DISCUSSION

You call this function to determine whether a sprite exists at a specified location in the sprite's display coordinate system. This function is useful for hit testing a subset of the sprites in a sprite world and for detecting multiple hits for a single location.

You should apply the sprite's matrix to the location before passing it to `SpriteHitTest`. To convert a location to local coordinates, you should use the `GlobalToLocal` function to convert the location to your window's local coordinate system and then apply the inverse of the sprite world's matrix to the location.

You can pass flags to this function to control the hit testing operation more precisely. For example, you may want the hit test operation to detect a sprite whose bounding box contains the specified location.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified

GetSpriteProperty

The GetSpriteProperty function retrieves the value of the specified sprite property.

```
pascal OSErr GetSpriteProperty (Sprite theSprite,
                    long propertyType,
                    void *propertyValue);
```

theSprite Specifies the sprite for this operation.

propertyType Specifies the property whose value should be retrieved.

propertyValue
On return, contains a pointer to a variable in which the property will be returned.

DISCUSSION

You call this function to retrieve a value of a sprite property. You set the propertyType parameter to the property you want to retrieve. The following table lists the sprite properties and the data types of the corresponding property values.

Sprite Property	Data Type
kSpritePropertyMatrix	MatrixRecord
kSpritePropertyImageDescription	ImageDescriptionHandle
kSpritePropertyImageDataPtr	Ptr
kSpritePropertyVisible	Boolean
kSpritePropertyLayer	short
kSpritePropertyGraphicsMode	ModifierTrackGraphicsModeRecord

In the case of the `kSpritePropertyImageDescription` and `kSpritePropertyImageDataPtr` properties, this function does not return a copy of the data; rather, the pointers returned are references to the sprite's data.

RESULT CODES

`noErr`	0	No error
`invalidSpritePropertyErr`	-2065	The specified sprite property does not exist

SetSpriteProperty

The `SetSpriteProperty` function sets the specified property of a sprite.

```
pascal OSErr SetSpriteProperty (Sprite theSprite,
                    long propertyType,
                    void *propertyValue);
```

`theSprite` Specifies the sprite for this operation.

`propertyType` Specifies the property to be set.

`propertyValue`
Specifies the new value of the property.

DISCUSSION

You animate a sprite by modifying its properties. You call this function to modify a property of a sprite. This function invalidates the sprite's sprite world as needed.

You set the `propertyType` parameter to the property you want to modify. Depending on the property type, you set the `propertyValue` parameter to either a pointer to the property value or the property value itself, cast as a `void*`. The

following table lists the sprite properties and the data types of the corresponding property values.

Sprite Property	Data Type
kSpritePropertyMatrix	MatrixRecord *
kSpritePropertyImageDescription	ImageDescriptionHandle
kSpritePropertyImageDataPtr	Ptr
kSpritePropertyVisible	Boolean
kSpritePropertyLayer	short
kSpritePropertyGraphicsMode	ModifierTrackGraphicsModeRecord *

RESULT CODES

noErr	0	No error
memFullErr	-108	Not enough memory available
invalidSpritePropertyErr	-2065	Specified sprite property does not exist

QT Atom Functions

This section describes the functions used to create and manipulate QT atom containers.

Creating and Modifying QT Atom Containers

QTNewAtomContainer

The `QTNewAtomContainer` function allows you to create a new atom container.

```
pascal OSErr QTNewAtomContainer (QTAtomContainer *atomData);
```

`atomData` Contains a pointer to an unallocated atom container data structure. On return, this parameter points to an allocated atom container.

DISCUSSION

This function creates a new, empty atom container structure. Once you have created an atom container, you can manipulate it using the atom container functions.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified
memFullError	-108	Not enough memory available

QTInsertChild

The QTInsertChild function creates a new child atom for the specified parent atom.

```
pascal OSErr QTInsertChild (QTAtomContainer container,
                    QTAtom parentAtom,
                    QTAtomType atomType,
                    QTAtomID id,
                    short index,
                    long dataSize,
                    void *data,
                    QTAtom *newAtom);
```

container	Specifies the atom container that contains the parent atom. The atom container must not be locked.
parentAtom	Specifies the parent atom within the atom container.
atomType	Specifies the type of the new atom to be inserted.
id	Specifies the ID of the new atom to be inserted. This ID must be unique among atoms of the same type for the specified parent. If you set this parameter to 0, this function will assign a unique ID to the atom.

`index`	Specifies the index of the new atom among atoms with the same parent. To insert the first atom for the specified parent, you should set the `index` parameter to 1. To insert an atom as the last atom in the child list, you should set the `index` parameter to 0.
`dataSize`	Specifies the size of the data for the new atom. If the new atom is to be a leaf atom or if you want to add the atom's data later, you should pass 0 for this parameter.
`data`	Contains a pointer to a buffer containing the data for the new atom. If you set the value of the `dataSize` parameter to 0, you should pass `nil` for this parameter.
`newAtom`	Contains a pointer to data of type `QTAtom`. On return, this parameter points to the newly created atom. You can pass `nil` for this parameter if you do not need a reference to the newly created atom.

DISCUSSION

You call this function to create a new child atom. The new child atom has the specified atom type and atom ID, and is inserted into its parent atom's child list at the specified index; any existing atoms at the same index or greater are moved toward the end of the child list. Index values greater than the index of the last atom in the child list plus 1 are invalid.

To create the new atom as a leaf atom that contains data, you should specify the data and its size using the `data` and `dataSize` parameters.

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified
`memLockedErr`	-117	Trying to move a locked block
`atomIndexInvalidErr`	-2104	Specifed index is out of range
`duplicateAtomTypeAndIDErr`	-2105	An atom with the same type and ID already exists for the specified parent

Memory Manager errors, as documented in *Inside Macintosh: Memory*.

QTInsertChildren

The `QTInsertChildren` function inserts a container of atoms as children of the specified parent atom.

```
pascal OSErr QTInsertChildren (QTAtomContainer container,
                    QTAtom parentAtom,
                    QTAtomContainer childrenContainer);
```

`container` Specifies the atom container that contains the parent atom. The atom container must not be locked.

`parentAtom` Specifies the parent atom within the atom container.

`childrenContainer`
Specifies the atom container that contains the child atoms to be inserted.

DISCUSSION

You call this function to insert a container of atoms as children of a parent atom in another atom container. Each child atom is inserted as the last atom of its type and is assigned a corresponding index. The ID of a child atom to be inserted must not duplicate that of an existing child atom of the same type.

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified
`memLockedErr`	-117	Trying to move a locked block
`duplicateAtomTypeAndIDErr`	-2105	An atom with the same type and ID already exists for the specified parent

Memory Manager errors, as documented in *Inside Macintosh: Memory*.

QTReplaceAtom

The `QTReplaceAtom` function replaces the contents of an atom and its children with a different atom and its children.

```
pascal OSErr QTReplaceAtom (QTAtomContainer targetContainer,
                    QTAtom targetAtom,
                    QTAtomContainer replacementContainer,
                    QTAtom replacementAtom);
```

`targetContainer`
Specifies the atom container that contains the atom to be replaced. The atom container must not be locked.

`targetAtom` Specifies the atom to be replaced.

`replacementContainer`
Specifies the atom container that contains the replacement atom.

`replacementAtom`
Specifies the replacement atom.

DISCUSSION

The target atom and the replacement atom must be of the same type. The target atom maintains its original atom ID. This function does not modify the replacement container.

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified
`memLockedErr`	-117	Trying to move a locked block
`atomsNotOfSameTypeErr`	-2103	The specified atoms are not of the same type

Memory Manager errors, as documented in *Inside Macintosh: Memory*.

QTSwapAtoms

The `QTSwapAtoms` function swaps the contents of two atoms in an atom container.

```
pascal OSErr QTSwapAtoms (QTAtomContainer container,
                          QTAtom atom1,
                          QTAtom atom2);
```

`container` Specifies the atom container for this operation.

`atom1` Specifies an atom to be swapped with the atom specified by `atom2`.

`atom2` Specifies an atom to be swapped with the atom specified by `atom1`.

DISCUSSION

You call this function to swap the contents of two atoms in an atom container. After swapping, the ID and index of each atom remains the same. The two atoms specified must be of the same type. Either atom may be a leaf atom or a container atom.

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified
`atomNotOfSameTypeErr`	-2103	The specified atoms are not of the same type

Memory Manager errors, as documented in *Inside Macintosh: Memory.*

QTSetAtomID

The `QTSetAtomID` function changes the ID of an atom.

```
pascal OSErr QTSetAtomID (QTAtomContainer container,
                          QTAtom atom,
                          QTAtomID newID);
```

container	Specifies the atom container for this operation.
atom	Specifies the atom to be modified.
newID	Specifies the new ID for the atom.

DISCUSSION

You cannot change an atom's ID to an ID already assigned to a sibling atom of the same type. Also, you cannot change the ID of the container itself by passing 0 for the atom parameter.

RESULT CODES

noErr	0	No error
duplicateAtomTypeAndIDErr	-2105	An atom with the same type and ID already exists for the specified parent
invalidAtomErr	-2106	Atom specified by container and offset does not exist, container may be invalid

QTSetAtomData

The QTSetAtomData function changes the data of a leaf atom.

```
pascal OSErr QTSetAtomData (QTAtomContainer container,
                    QTAtom atom,
                    long dataSize,
                    void *atomData);
```

container	Specifies the atom container that contains the atom to be modified.
atom	Specifies the atom to be modified.
dataSize	Specifies the length, in bytes, of the data pointed to by the atomData parameter.
atomData	Contains a pointer to the new data for the atom.

DISCUSSION

You call this function to replace a leaf atom's data with new data. Only leaf atoms contain data; this function returns an error if you pass it to a non-leaf atom.

The atom container specified by the `container` parameter should not be locked. This function may move memory; if the pointer specified by the `atomData` parameter is a dereferenced handle, you should lock the handle.

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified
`memLockedErr`	-117	Trying to move a locked block
`notLeafAtomErr`	-2102	Atom specifed by container and offset is not a leaf atom

Memory Manager errors, as documented in *Inside Macintosh: Memory.*

QTCopyAtom

The `QTCopyAtom` function copies an atom and its children to a new atom container.

```
pascal OSErr QTCopyAtom (QTAtomContainer container,
                         QTAtom atom,
                         QTAtomContainer *targetContainer);
```

`container` Specifies the atom container that contains the atom to be copied.

`atom` Specifies the atom to be copied.

`targetContainer`
Contains a pointer to an uninitialized atom container data structure. On return, this parameter points to an atom container that contains a copy of the atom.

DISCUSSION

To duplicate the entire container specified by the container parameter, you should pass a value of `kParentAtomIsContainer` for the `atom` parameter. The

caller is responsible for disposing of the new atom container by calling the `QTDisposeAtomContainer` function.

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified

Memory Manager errors, as documented in *Inside Macintosh: Memory*.

QTLockContainer

The `QTLockContainer` function locks an atom container in memory.

```
pascal OSErr QTLockContainer  (QTAtomContainer container);
```

`container` Specifies the atom container to be locked.

DISCUSSION

You should call this function to lock an atom container before calling `QTGetAtomDataPtr` to directly access a leaf atom's data. When you have finished accessing a leaf atom's data, you should call the `QTUnlockContainer` function.

You may make nested pairs of calls to `QTLockContainer` and `QTUnlockContainer`; you do not need to check the current state of the container first.

RESULT CODES

noErr	0	No error
invalidAtomContainerErr	-2107	Specified atom container is invalid

QTGetAtomDataPtr

The QTGetAtomDataPtr function retrieves a pointer to the atom data for the specified leaf atom.

```
pascal OSErr QTGetAtomDataPtr (QTAtomContainer container,
                    QTAtom atom,
                    long *dataSize,
                    Ptr *atomData);
```

container	Specifies the atom container that contains the leaf atom.
atom	Specifies the leaf atom whose data should be retrieved.
dataSize	On return, contains a pointer to the length, in bytes, of the leaf atom's data.
atomData	On return, contains a pointer to the leaf atom's data.

DISCUSSION

You call this function in retrieve a pointer to a leaf atom's data so that you can access the data directly. To ensure that the pointer returned in the atomData parameter will remain valid if memory is moved, you should call QTLockContainer before you call this function. If you do call QTLockContainer, you should call QTUnlockContainer when you have finished using the atomData pointer; if you pass a locked atom container to a function that resizes atom containers, the function will return an error.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified
notLeafAtomErr	-2102	Atom specifed by container and offset is not a leaf atom

QTUnlockContainer

The QTLockContainer function unlocks an atom container in memory.

```
pascal OSErr QTUnlockContainer (QTAtomContainer container);
```

container Specifies the atom container to be unlocked.

DISCUSSION

You should call this function to unlock an atom container when you have finished accessing a leaf atom's data.

You may make nested pairs of calls to QTLockContainer and QTUnlockContainer; you do not need to check the current state of the container first.

RESULT CODES

noErr	0	No error
invalidAtomContainerErr	-2107	Specified atom container is invalid

QTRemoveAtom

The QTRemoveAtom function removes an atom and its children from the specified atom container.

```
pascal OSErr QTRemoveAtom (QTAtomContainer container,
                           QTAtom atom);
```

container Specifies the atom container for this operation. The atom container must not be locked.

atom	Specifies the atom to be removed from the container.

DISCUSSION

You call this function to remove a particular atom and its children from an atom container. To remove all the atoms in an atom container, you should use the QTRemoveChildren function.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified
memLockedErr	-117	Trying to move a locked block

Memory Manager errors, as documented in *Inside Macintosh: Memory.*

QTRemoveChildren

The QTRemoveChildren function removes all the children of an atom from the specified atom container.

```
pascal OSErr QTRemoveChildren (QTAtomContainer container,
                               QTAtom atom);
```

container	Specifies the atom container for this operation. The atom container must not be locked.
atom	Specifies the atom whose children should be removed.

DISCUSSION

To remove all the atoms in the atom container, pass a value of kParentAtomIsContainer for the atom parameter.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified
memLockedErr	-117	Trying to move a locked block

Memory Manager errors, as documented in *Inside Macintosh: Memory*.

QTDisposeAtomContainer

The QTDisposeAtomContainer function disposes of an atom container.

```
pascal OSErr QTDisposeAtomContainer (QTAtomContainer atomData);
```

atomData Specifies the atom container to be disposed of.

DISCUSSION

You can call this function to dispose of an atom container data structure that was created by QTNewAtomContainer or QTCopyAtom.

RESULT CODES

noErr	0	No error
invalidAtomContainerErr	-2107	Specified atom container is invalid

Retrieving Atoms and Atom Data

QTGetNextChildType

The QTGetNextChildType function returns the next atom type in the child list of the specified parent atom.

```
pascal QTAtomType QTGetNextChildType (QTAtomContainer container,
                    QTAtom parentAtom,
                    QTAtomType currentChildType);
```

`container`	Specifies the atom container that contains the parent atom.
`parentAtom`	Specifies the parent atom for this operation.
`currentChildType`	Specifies the last atom type retrieved by this function.
function result	The atom type.

DISCUSSION

You can call this function to iterate through the atom types in a parent atom's child list. To retrieve the first atom type, you should set the value of the `currentChildType` parameter to 0. To retrieve subsequent atom types, you should set the value of the `currentChildType` parameter to the atom type retrieved by the previous call to this function.

QTCountChildrenOfType

The `QTCountChildrenOfType` function returns the number of atoms of a given type in the child list of the specified parent atom.

```
pascal short QTCountChildrenOfType (QTAtomContainer container,
                    QTAtom parentAtom,
                    QTAtomType childType);
```

`container`	Specifies the atom container that contains the parent atom.
`parentAtom`	Specifies the parent atom for this operation.
`childType`	Specifies the atom type for this operation.
function result	The number of atoms of the specified type in the parent atom's child list.

DISCUSSION

You can call this function to determine the number of atoms of a specified type in a parent atom's child list. To retrieve the total number of atoms in the child list, you should set the value of the `childType` parameter to 0. If the total

number of atoms in the parent atom's child list is 0, the parent atom is a leaf atom.

QTFindChildByIndex

The `QTFindChildByIndex` function retrieves an atom by index from the child list of the specified parent atom.

```
pascal QTAtom QTFindChildByIndex (QTAtomContainer container,
                    QTAtom parentAtom,
                    QTAtomType atomType,
                    short index,
                    QTAtomID *id);
```

`container`	Specifies the atom container that contains the parent atom.
`parentAtom`	Specifies the parent atom for this operation.
`atomType`	Specifies the type of the atom to be retrieved.
`index`	Specifies the index of the atom to be retrieved.
`id`	Contains a pointer to an uninitialized `QTAtomID` data structure. On return, if the atom specified by `index` was found, the `QTAtomID` data structure contains the atom's ID.
function result	The child atom, if found; otherwise, 0.

DISCUSSION

You call this function to search for and retrieve an atom by its type and index within that type from a parent atom's child list. If you do not want this function to return the atom's ID, set the value of the `id` parameter to `nil`.

QTFindChildByID

The `QTFindChildByID` function retrieves an atom by ID from the child list of the specified parent atom.

```
pascal QTAtom QTFindChildByID (QTAtomContainer container,
                    QTAtom parentAtom,
                    QTAtomType atomType,
                    QTAtomID id,
                    short *index);
```

`container`	Specifies the atom container that contains the parent atom.
`parentAtom`	Specifies the parent atom for this operation.
`atomType`	Specifies the type of the atom to be retrieved.
`id`	Specifies the ID of the atom to be retrieved.
`index`	Contains a pointer to an uninitialized short integer. On return, if the atom specified by `id` was found, the integer contains the atom's index.
function result	The child atom, if found; otherwise, 0.

DISCUSSION

You call this function to search for and retrieve an atom by its type and ID from a parent atom's child list. If you do not want this function to return the atom's index, set the value of the `index` parameter to `nil`.

QTNextChildAnyType

The `QTNextChildAnyType` function returns the next atom in the child list of the specified parent atom.

```
pascal OSErr QTNextChildAnyType (QTAtomContainer container,
                    QTAtom parentAtom,
                    QTAtom currentChild,
                    QTAtom *nextChild);
```

container	Specifies the atom container that contains the parent atom.
parentAtom	Specifies the parent atom for this operation.
currentChild	Specifies the last atom retrieved by this function.
nextChild	Contains a pointer to an uninitialized QTAtom data structure. On return, the data structure contains the offset of the next atom in the child list after the atom specified by currentChild, or 0 if the atom specified by currentChild was the last atom in the list.

DISCUSSION

You can call this function to iterate through all the atoms in a parent atom's child list, regardless of their types and IDs. To retrieve the first atom in the child list, set the value of the currentChild parameter to 0.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified
invalidAtomErr	-2106	Atom specified by container and offset does not exist, container may be invalid

QTCopyAtomDataToHandle

The QTCopyAtomDataToHandle function copies the specified leaf atom's data to a handle.

```
pascal OSErr QTCopyAtomDataToHandle (QTAtomContainer container,
                    QTAtom atom,
                    Handle targetHandle);
```

container	Specifies the atom container that contains the leaf atom.
atom	Specifies the leaf atom whose data should be copied.
targetHandle	Contains a handle. On return, the handle contains the atom's data. The handle must not be locked.

DISCUSSION

You call this function, passing an initialized handle, to retrieve a copy of a leaf atom's data. This function resizes the handle, if necessary.

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified
`memLockedErr`	-117	Trying to move a locked block
`notLeafAtomErr`	-2102	Atom specifed by container and offset is not a leaf atom

Memory Manager errors, as documented in *Inside Macintosh: Memory.*

QTCopyAtomDataToPtr

The `QTCopyAtomDataToPtr` function copies the specified leaf atom's data to a buffer.

```
pascal OSErr QTCopyAtomDataToPtr (QTAtomContainer container,
                    QTAtom atom,
                    Boolean sizeOrLessOK,
                    long size,
                    void *targetPtr,
                    long *actualSize);
```

`container`	Specifies the atom container that contains the leaf atom.
`atom`	Specifies the leaf atom whose data should be copied.
`sizeOrLessOK`	Specifies whether the function may copy fewer bytes than the number of bytes specified by the `size` parameter.
`size`	Specifies the length, in bytes, of the buffer pointed to by the `targetPtr` parameter.
`targetPtr`	Contains a pointer to a buffer. On return, the buffer contains the atom data.
`actualSize`	Contains a pointer to a long integer which, on return, contains the number of bytes copied to the buffer.

DISCUSSION

You call this function, passing a data buffer, to retrieve a copy of a leaf atom's data. The buffer must be large enough to contain the atom's data. The buffer may be larger than the amount of atom data if you set the value of the `sizeOrLessOK` parameter to `true`. You can determine the size of an atom's data by calling `QTGetAtomDataPtr`.

This function may move memory.

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified
`notLeafAtomErr`	-2102	Atom specifed by container and offset is not a leaf atom

Memory Manager errors, as documented in *Inside Macintosh: Memory*.

QTGetAtomTypeAndID

The `QTGetAtomTypeAndID` function retrieves an atom's type and ID.

```
pascal OSErr QTGetAtomTypeAndID (QTAtomContainer container,
                    QTAtom atom,
                    QTAtomType *atomType,
                    QTAtomID *id);
```

`container`	Specifies the atom container that contains the atom.
`atom`	Specifies the atom whose type and ID should be retrieved.
`atomType`	Contains a pointer to an atom type. On return, this parameter points to the type of the specified atom. You can pass `nil` for this parameter if you do not need this information.
`id`	Contains a pointer to an atom ID. On return, this parameter points to the ID of the specified atom. You can pass `nil` for this parameter if you do not need this information.

DISCUSSION

You call this function to retrieve the type and ID for a particular atom.

RESULT CODES

`noErr`	0	No error
`invalidAtomErr`	-2106	Atom specified by container and offset does not exist, container may be invalid

Component Manager

Contents

This chapter discusses changes to the Component Manager as documented in Chapter 6 of *Inside Macintosh: More Macintosh Toolbox*.

New Features of the Component Manager

QuickTime 2.5 adds five new functions to the Component Manager: `GetComponentTypeModSeed`, `OpenAComponent`, `OpenADefaultComponent`, `OpenAComponentResFile`, and `CallComponentFunctionWithStorageProcInfo`.

The `GetComponentTypeModSeed` function is similar to the `GetComponentListModSeed` function. The `OpenAComponent`, `OpenADefaultComponent` and `OpenAComponentResFile` functions expand upon existing Component Manager routines by adding an error return value. The `CallComponentFunctionWithStorageProcInfo` function is used for PowerPC-native component dispatching.

The Component Manager has been enhanced in QuickTime 2.5 to better support PowerPC-native components. For more details, please refer to *Component Manager Reference* and *Native Component Manager*.

PowerPC-Native Component Manager Support

The Component Manager has been enhanced in QuickTime 2.5 to better support PowerPC-native components.

The component manager dispatcher for calling components is now fat, avoiding the overhead of the mixed mode switches through the old 68K Component Manager dispatch for native-native component calls. `DelegateComponentCall` is also fat, so native-native delegations avoid the mixed mode switch.

In addition, writing the component dispatch routine for native components has been made significantly easier with the addition of `CallComponentFunctionWithStorageProcInfo`. This call is analagous to `CallComponentFunctionWithStorage`, with the addition of a parameter to pass the proc info for the desired function. This allows the Mixed Mode manager to

correctly dispatch the call without your code having to unravel the parameters from the `ComponentParameters` block yourself.

To use the new `CallComponentFunctionWithStorageProcInfo` call, your component will need to link with `ComponentsInterfacesLib`.

An example component that uses this technique follows. The component supports the required suite of Open, Close, CanDo, and Version calls, as well as a Beep call (selector 0). The `ExampleComponentDispatch` and `ExampleFindRoutineProcPtr` routines provide the dispatching using the new `CallComponentFunctionWithStorageProcInfo` call.

```
#include <Sound.h>
#include <Components.h>

pascal ComponentResult ExampleComponentDispatch
        (ComponentParameters *params, Handle storage);

static ProcPtr ExampleFindRoutineProcPtr
        (short selector, ProcInfoType *procInfo);

pascal ComponentResult
        ExampleCanDo(Handle storage, short selector);

pascal ComponentResult
        ExampleOpen(Handle storage, ComponentInstance self);

pascal ComponentResult
        ExampleClose(Handle storage,ComponentInstance self);

pascal ComponentResult
        ExampleVersion(Handle storage);

pascal ComponentResult
        ExampleBeep(Handle storage);

#ifdef GENERATINGPOWERPC

struct RoutineDescriptor ExampleComponentDispatchRD =
        BUILD_ROUTINE_DESCRIPTOR(
            (kPascalStackBased | RESULT_SIZE (kFourByteCode) |
                STACK_ROUTINE_PARAMETER (1, kFourByteCode) |
```

```
                STACK_ROUTINE_PARAMETER (2, kFourByteCode)),
            ExampleComponentDispatch);
#endif

enum {
    kExampleBeepSelect = 0
};

enum {
    uppExampleBeepProcInfo = kPascalStackBased
         | RESULT_SIZE(SIZE_CODE(sizeof(ComponentResult)))
         | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(Handle)))
};

pascal ComponentResult
ExampleComponentDispatch(ComponentParameters *params, Handle storage)
{
    ProcPtr theProc;
    ProcInfoType theProcInfo;
    ComponentResult result = codecUnimpErr;
    theProc = ExampleFindRoutineProcPtr(params->what, &theProcInfo);
    if (theProc)
        result = CallComponentFunctionWithStorageProcInfo(
                (Handle)storage, params, theProc, theProcInfo);
    return result;
}

static ProcPtr
ExampleFindRoutineProcPtr(short selector, ProcInfoType *procInfo)
{
    ProcPtr aProc;
    ProcInfoType pi;

#define ComponentCall(a)\
        case kComponent##a##Select:  \
            aProc = (ProcPtr)Example##a; \
            pi = uppCallComponent##a##ProcInfo; break;

#define ExampleCall(a)\
        case kExample##a##Select: \
            aProc = (ProcPtr)Example##a; \
```

```
            pi = uppExample##a##ProcInfo; break;
    switch (selector) {
        ComponentCall(Version)
        ComponentCall(CanDo)
        ComponentCall(Close)
        ComponentCall(Open)
        ExampleCall(Beep)

        default:
            aProc = nil;
            pi = 0;
        }
    *procInfo = pi;
    return aProc;
}

pascal ComponentResult ExampleCanDo(Handle storage, short selector)
{
    ProcInfoType ignoreResult;
    return (ExampleFindRoutineProcPtr(selector,&ignoreResult) != 0);
}

pascal ComponentResult ExampleOpen(Handle storage,
    ComponentInstance self)
{
    return noErr;
}

pascal ComponentResult ExampleClose(Handle storage,
    ComponentInstance self)
{
    return(noErr);
}

pascal ComponentResult ExampleVersion(Handle storage)
{
    return 0x00010001;
}
```

```
pascal ComponentResult ExampleBeep(Handle storage)
{
    SysBeep(2);
    return noErr;
}
```

Component Manager Reference

Dispatching to Component Routines

CallComponentFunctionWithStorageProcInfo

To use the new `CallComponentFunctionWithStorageProcInfo` call, your component will need to link with `ComponentsInterfacesLib`.

```
pascal long CallComponentFunctionWithStorageProcInfo(
                    Handle storage,
                    ComponentParameters *params,
                    ProcPtr func,
                    long funcProcInfo);
```

`storage` A handle to the memory associated with the current connection. The Component Manager provides this handle to your component along with the request.

`paams` The component parameters record that your component received from the Component Manager.

`func` The address of the function that is to handle the request. The Component Manager calls the routine referred to by the func parameter as a Pascal function with the parameters that were originally provided by the application. These parameters are preceded by a handle to the memory associated with the current connection. The routine referred to by the func

parameter must return a function result of type ComponentResult (a long integer) indicating the success or the failure of the operation.

Note that for PowerPC code, the func parameter should still point to the routine itself—not to a `RoutineDescriptor` or Universal Procedure Pointer.

`funcProcInfo` The procedure information for the routine referred to by the func parameter. See *Inside Macintosh: PowerPC System Software*, pages 2-14 through 2-21 for information on procedure information data.

Finding Components

GetComponentTypeModSeed

The `GetComponentTypeModSeed` function allows you to determine if the specified type of registered component has changed. This function returns the value of the component registration seed number for the specified component type. By comparing this value to values previously returned by this function, you can determine whether the component registry for the specified type has changed.

```
pascal long GetComponentTypeModSeed (OSType componentType);
```

`componentType`

A four-character code that identifies the type of component. All components of a particular type support a common set of interface routines. Your application uses this field to search for components of a given type.

function result

Returns a long integer containing the component registration seed number. Each time the Component Manager registers or unregisters a component, it generates a new, unique seed number.

DISCUSSION

This function is similar to the `GetComponentListModSeed` function. Unlike `GetComponentListModSeed`, the `GetComponentTypeModSeed` function is specific to a single component type. This allows you to know if, for example, the registration of image decompressor (‘`imdc`’) components has changed regardless of other component changes.

Opening and Closing Components

OpenAComponent

The `OpenAComponent` function is similar to `OpenComponent`, except that its return value is an `OSErr`. The `ComponentInstance` of the newly opened component is passed back through the `ci` argument.

```
pascal OSErr OpenAComponent (
                    Component aComponent,
                    ComponentInstance *ci);
```

`aComponent` A component identifier that specifies the component to open. Your application obtains this identifier from the `FindNextComponent` function. If your application registers a component, it can also obtain a component identifier from the `RegisterComponent` or `RegisterComponentResource` function.

`ci` A pointer to a field to receive the `ComponentInstance` of the newly-opened component.

OpenADefaultComponent

The `OpenADefaultComponent` function is similar to `OpenDefaultComponent`, except that its return value is an `OSErr`. The `ComponentInstance` of the newly opened component is passed back through the `ci` argument.

```
pascal OSErr OpenADefaultComponent (
                    OSType componentType,
                    OSType componentSubType,
                    ComponentInstance *ci);
```

`componentType`
: A four-character code that identifies the type of component. All components of a particular type support a common set of interface routines. Your application uses this field to search for components of a given type.

`componentSubType`
: A four-character code that identifies the subtype of the component. Different subtypes of a component type may support additional features or provide interfaces that extend beyond the standard routines for a given component type. For example, the subtype of an image compressor component indicates the compression algorithm employed by the compressor.

  Your application can use the `componentSubType` field to perform a more specific lookup operation than is possible using only the `componentType` field. For example, you may want your application to use only components of a certain component type (‘`draw`’) that also have a specific subtype (‘`oval`’). Set this parameter to 0 to select a component with any subtype value.

`ci`
: A pointer to a field to receive the `ComponentInstance` of the newly-opened component.

Accessing a Component's Resource File

OpenAComponentResFile

The `OpenADefaultComponent` function is similar is similar to `OpenComponentResFile`, except that its return value is an `OSErr`. The resource reference number of the newly opened component file is passed back through the `resRef` argument.

```
pascal OSErr OpenAComponentResFile (
                    Component aComponent,
                    short *resRef);
```

`aComponent` A component identifier that specifies the component whose resource file you want to open. Your application can obtain this identifier from the `RegisterComponentResource` or `FindNextComponent` functions, or it can be a `ComponentInstance` (in which case it's the result of `OpenAComponent` or `OpenADefaultComponent`).

`resRef` A pointer to a field to receive the resource reference number of the newly opened component resource file.

Image Compression Manager

Contents

This chapter discusses new features and changes to the Image Compression Manager as documented in Chapter 3 of *Inside Macintosh: QuickTime*.

New Features of the Image Compression Manager

ColorSync Support

ColorSync is a system extension that provides a platform for consistent color reproduction between widely varying output devices. ColorSync color matching capability was added to the Image Compression Manager picture drawing functions in QuickTime 1.6.1. You can now accurately reproduce color images (not movies) with the `DrawPicture` functions by setting the `useColorMatching` flag in the flags parameter to these functions.

```
enum {
    useColorMatching    = 4
};
```

For more information about QuickTime picture drawing functions, see "Working With Pictures and PICT Files," beginning on page 3-88 of *Inside Macintosh: QuickTime*.

Asynchronous Decompression

QuickTime 2.0 introduced the concept of scheduled asynchronous decompression operations. Decompressor components can allow applications to queue decompression operations and specify when those operations should take place. The Image Compression Manager provides a new function, `DecompressSequenceFrameWhen` (page 3-13), that allows applications to schedule an asynchronous decompression operation.

Timecode Support

Timecode tracks were introduced in QuickTime 2.0. The Image Compression Manager and compressor components have been enhanced to support timecode information. The Image Compression Manager `SetDSequenceTimeCode`

function (page 3-20) allows you to set the timecode value for a frame that is to be decompressed. For more information about timecode tracks and the timecode media handler, see Chapter 1, "Movie Toolbox."

Data Source Support

QuickTime 2.1 introduced support for an arbitrary number of sources of data for an image sequence. This functionality forms the basis for dynamically modifying parameters to a decompressor. It also allows for codecs to act as special effects components, providing filtering and transition type effects. A client can attach an arbitrary number of additional inputs to the codec. It is up to the particular codec to determine whether to use each input and how to interpret the input. For example, an 8-bit gray image could be interpreted as a blend mask or as a replacement for one of the RGB data planes.

To create a new data source, use the `CDSequenceNewDataSource` function (page 3-25).

Working with Alpha Channels

QuickTime has always supported compressing and storing images with an alpha channel. In QuickTime 2.5, the Image Compression Manager has been updated to support using the alpha channel when displaying images. Alpha channels are supported only for 32-bit images. The high byte of each pixel contains the alpha channel. The alpha channel can be interpreted in one of three ways:

- straight alpha
- pre-multiplied with white
- pre-multiplied with black

QuickTime uses the alpha channel to define how an image is to be combined with the image that is already present at the location to which it will be drawing. This is similar to how QuickDraw's blend mode works. To combine an image containing an alpha channel with another image, you specify how the alpha channel should be interpreted by specifying one of the new alpha channel graphics modes defined by QuickTime.

Straight alpha means that the color components of each pixel should be combined with the corresponding background pixel based on the value contained in the alpha channel. For example, if the alpha value is 0, only the background pixel

will appear. If the alpha value is 255, only the foreground pixel will appear. If the alpha value is 127, then (127/255) of the foreground pixel will be blended with (128/255) of the background pixel to create the resulting pixel, and so on.

Pre-multiplied with white means that the color components of each pixel have already been blended with a white pixel, based on their alpha channel value. Effectively, this means that the image has already been combined with a white background. To combine the image with a different background color, QuickTime must first remove the white from each pixel and then blend the image with the actual background pixels. Images are often pre-multipled with white as this reduces the appearance of jagged edges around objects.

Pre-multipled with black is the same as pre-multipled with white, except the background color that the image has been blended with is black instead of white.

Note
Although you pass these new alpha channel graphics modes to QuickTime in the same way as you would traditional QuickDraw transfer modes, these modes are not supported by QuickDraw and will cause unpredictable results if passed to QuickDraw routines. ◆

The Image Compression Manager defines the following constants for specifying alpha channel graphics modes:

```
enum {
    graphicsModeStraightAlpha       = 256,
    graphicsModePreWhiteAlpha       = 257,
    graphicsModePreBlackAlpha       = 258
    graphicsModeStraightAlphaBlend  = 260
};
```

The `graphicsModeStraightAlpha`, `graphicsModePreWhiteAlpha`, and `graphicsModePreBlackAlpha` graphics modes cause QuickTime to draw the image interpreting the alpha channel as specified. The graphics mode `graphicsModeStraightAlphaBlend` causes QuickTime to interpret the alpha channel as a straight alpha channel, but when it draws, combines the pixels together and applies the `opColor` supplied with the graphics mode to the alpha channel. This provides an easy way to combine images using both an alpha channel and a blend level. This can be useful when compositing 3D rendered images over video.

To draw a compressed image containing an alpha channel, that image must be compressed using an image compression format that is capable of storing the alpha channel information. The Animation, Planar RGB and None compressors store alpha channel data in the "Millions of Colors +" (32-bit) mode.

You use the `MediaSetGraphicsMode` function to set a movie track to use an alpha channel graphics mode. You use the `SetDSequenceTransferMode` function to set an image sequence to use an alpha channel graphics mode.

Working With Video Fields

QuickTime 2.5 introduces support for working directly with fields of interlaced video, such as those created by some motion JPEG compressors.

Because video processing applications sometimes need to perform operations on individual fields (for example, reversing them or combining one field of a frame with a field from another frame), QuickTime now provides a method for accessing the individual fields without having to decompress them first. Previously such operations required decompressing each frame, copying the appropriate fields, and then recompressing. This was a time consuming process that could result in a loss of image quality due to the decompression and recompression of the video data.

Three new functions (`ImageFieldSequenceBegin`, `ImageFieldSequenceExtractCombine`, and `ImageFieldSequenceEnd`) allow an application to request that field operations be performed directly on the compressed data. These functions accept one or two compressed images as input and create a single compressed image on output.

The Apple Component Video and Motion JPEG compressors support image field functions in QuickTime 2.5. See the description of the `ImageFieldSequenceBegin`, `ImageFieldSequenceExtractCombine`, and `ImageFieldSequenceEnd` functions for information on how to process image fields in your application. See Chapter 4, "Image Compressor Components," for information on incorporating support for these functions in other compressors.

Packetization Information

QuickTime video compressors are increasingly being used for videoconferencing applications. Image data from a compressor is typically split into network-packet-sized pieces, transmitted through a packet-based protocol

(such as UDP or DDP), and reassembled into a frame by the receiver(s). Typically, a lost packet causes an entire frame to be dropped; without all the data for a given frame, the decompressor cannot decode the image. When the loss of one packet forces others to be unusable, the loss rate is effectively multiplied by a large factor.

Some compression methods, however, such as H.261, can divide a compressed image into pieces which can be decoded independently. Some videoconferencing protocols, such as the Internet's Real Time Protocol (RTP, RFC#1889), specify that data compressed using H.261 must be packetized into independently decodable chunks. While RTP demands this packetization information from the compressor, other protocols, such as QuickTime Conferencing's MovieTalk protocol, can optionally use this information to effectively reduce loss rates.

QuickTime 2.5 provides four new functions to support packetization: `SetCSequencePreferredPacketSize` (page 3-32), `SGSetPreferredPacketSize` (page 8-5), `SGGetPreferredPacketSize` (page 8-6), and `VDSetPreferredPacketSize` (page 9-7). In addition, the `CodecCompressParams` structure (page 4-14) includes a new field, `preferredPacketSizeInBytes`. See Chapter 4, "Image Compressor Components," for information about supporting packetization in image compressor components.

For application developers, the important function is `SGSetPreferredPacketSize`, which is described in Chapter 8, "Sequence Grabber Channel Components." The `SetCSequencePreferredPacketSize` function is described later in this chapter. For information about the `VDSetPreferredPacketSize` function, see Chapter 9, "Video Digitizer Components."

Using the Image Compression Manager

Using Screen Buffers and Image Buffers

In QuickTime 2.1, support for screen buffers was removed. All requests for screen buffers are now converted into requests for image buffers. Applications should no longer request screen buffers.

Image Compression Manager Reference

Data Types

Image Compression Manager Function Control Flags

This section describes the new function control flags provided by the Image Compression Manager.

```
enum {
    codecFlagDontUseNewImageBuffer      = (1L << 10),
    codecFlagInterlaceUpdate            = (1L << 11),
    codecFlagCatchUpDiff                = (1L << 12)
};
```

Flag descriptions

`codecFlagDontUseNewImageBuffer`
Forces an error to be returned when a new image buffer would have to be allocated instead of allocating the new buffer.

`codecFlagInterlaceUpdate`
Updates the screen interlacing even and odd scan lines to reduce tearing artifacts (if the decompressor supports this mode).

`codecFlagCatchUpDiff`
Notifies the codec that the currently displayed frame is being displayed late in an attempt to "catch up" to the current frame, which only happens with compression formats that support frame differencing, You can pass this flag to any of the `DecompressSequenceFrame` calls.

Constants

This section describes the new constants provided by the Image Compression Manager.

```
/* alpha channel graphics modes */
enum {
    graphicsModeStraightAlpha       = 256,
    graphicsModePreWhiteAlpha       = 257,
    graphicsModePreBlackAlpha       = 258,
    graphicsModeStraightAlphaBlend  = 260
};

/* fieldFlags for the ImageFieldSequenceExtractCombine function */
enum {
    evenField1ToEvenFieldOut        = 1<<0,
    evenField1ToOddFieldOut         = 1<<1,
    oddField1ToEvenFieldOut         = 1<<2,
    oddField1ToOddFieldOut          = 1<<3,
    evenField2ToEvenFieldOut        = 1<<4,
    evenField2ToOddFieldOut         = 1<<5,
    oddField2ToEvenFieldOut         = 1<<6,
    oddField2ToOddFieldOut          = 1<<7
};
```

Functions

Working With Sequences

DecompressSequenceBeginS

Sends a sample image to a decompressor.

```
pascal OSErr DecompressSequenceBeginS (
                    ImageSequence *seqID,
                    ImageDescriptionHandle desc,
                    Ptr data,
                    long dataSize,
                    CGrafPtr port,
                    GDHandle gdh,
                    const Rect *srcRect,
                    MatrixRecordPtr matrix,
                    short mode,
                    RgnHandle mask,
                    CodecFlags flags,
                    CodecQ accuracy,
                    DecompressorComponent codec);
```

`seqID` Contains a pointer to a field to receive the unique identifier for this sequence returned by the `CompressSequenceBegin` function.

`desc` Contains a handle to the image description structure that describes the compressed image.

`data` Points to the compressed image data. This pointer must contain a 32-bit clean address. If you use a dereferenced, locked handle, you must call the Memory Manager's `StripAddress` function before you use that pointer with this parameter.

`dataSize` Specifies the size of the `data` buffer.

`port` Points to the graphics port for the destination image.

gdh	Contains a handle to the graphics device record for the destination image.
srcRect	Contains a pointer to a rectangle defining the portions of the image to decompress.
matrix	Points to a matrix structure that specifies how to transform the image during decompression.
mode	Specifies the transfer mode for the operation.
mask	Contains a handle to the clipping region in the destination coordinate system.
flags	Contains flags providing further control information.
accuracy	Specifies the accuracy desired in the decompressed image.
codec	Contains the compressor identifier.

DISCUSSION

The `DecompressSequenceBeginS` function, introduced in QuickTime 1.6.1, allows you to pass a compressed sample so the codec can perform preflighting before the first `DecompressSequenceFrame` call.

SetSequenceProgressProc

Installs a progress procedure for a sequence.

```
pascal OSErr SetSequenceProgressProc (
                    ImageSequence seqID,
                    ICMProgressProcRecord *progressProc);
```

seqID	Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.
progressProc	Points to a record containing information about the application's progress procedure.

DISCUSSION

The `SetSequenceProgressProc` function, introduced in QuickTime 1.6.1, allows you to set a progress procedure on a compression or decompression sequence, just as earlier versions of QuickTime allowed you to set a progress procedure when compressing or decompressing a still image.

GetCSequenceMaxCompressionSize

The `GetCSequenceMaxCompressionSize` function allows your application to determine the maximum size an image will be after compression for a given compression sequence. You must have already creating a compression sequence with `CompressSequenceBegin`.

```
pascal OSErr GetCSequenceMaxCompressionSize(
                    ImageSequence seqID,
                    PixMapHandle src,
                    long *size);
```

`seqID` Contains the unique sequence identifier that was returned by the `CompressSequenceBegin` function.

`src` Contains a handle to the source pixel map. The compressor uses only the image's size and pixel depth to determine the maximum size of the compressed image.

`size` Contains a pointer to a field to receive the maximum size, in bytes, of the compressed image.

DISCUSSION

The `GetCSequenceMaxCompressionSize` function is similar to the GetMaxCompressionSize function, but operates on a compression sequence instead of requiring the application to pass the individual parameters about the source image.

DecompressSequenceFrameWhen

Queues a frame for decompression and specifies the time at which decompression will begin.

```
pascal OSErr DecompressSequenceFrameWhen (
                    ImageSequence seqID,
                    Ptr data,
                    long dataSize,
                    CodecFlags inFlags,
                    CodecFlags *outFlags,
                    ICMCompletionProcRecordPtr asyncCompletionProc,
                    const ICMFrameTimeRecord *frameTime);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.

`data` Points to the compressed image data. This pointer must contain a 32-bit clean address. If you use a dereferenced, locked handle, you must call the Memory Manager's `StripAddress` function before you use that pointer with this parameter.

`dataSize` Specifies the size of the `data` buffer.

`inFlags` Contains flags providing further control information. See *Inside Macintosh: QuickTime* for information about `CodecFlags` fields. The following flags are valid for this function:

`codecFlagNoScreenUpdate`
Controls whether the decompressor updates the screen image. If you set this flag to 1, the decompressor does not write the current frame to the screen, but does write the frame to its offscreen image buffer (if one was allocated). If you set this flag to 0, the decompressor writes the frame to the screen.

`codecFlagDontOffscreen`
Controls whether the decompressor uses the offscreen buffer during sequence decompression. This flag is only used with sequences that have been temporally compressed. If this flag is set to 1, the

decompressor does not use the offscreen buffer during decompression. Instead, the decompressor returns an error. This allows your application to refill the offscreen buffer. If this flag is set to 0, the decompressor uses the offscreen buffer if appropriate.

`codecFlagOnlyScreenUpdate`
Controls whether the decompressor decompresses the current frame. If you set this flag to 1, the decompressor writes the contents of its offscreen image buffer to the screen, but does decompress the current frame. If you set this flag to 0, the decompressor decompresses the current frame and writes it to the screen. You can set this flag to 1 only if you have allocated an offscreen image buffer for use by the decompressor.

`outFlags` Contains status flags. The decompressor updates these flags at the end of the decompression operation. See *Inside Macintosh: QuickTime* for information about `CodecFlags` constants. The following flags may be set by this function:

`codecFlagUsedNewImageBuffer`
Indicates to your application that the decompressor used the offscreen image buffer for the first time when it processed this frame. If this flag is set to 1, the decompressor used the image buffer for this frame and this is the first time the decompressor used the image buffer in this sequence.

`codecFlagUsedImageBuffer`
Indicates whether the decompressor used the offscreen image buffer. If the decompressor used the image buffer during the decompress operation, it sets this flag to 1. Otherwise, it sets this flag to 0.

codecFlagDontUseNewImageBuffer

This input flag forces an error to be returned when a new image buffer would have to be allocated instead of allocating the new buffer.

codecFlagInterlaceUpdate

This input flag updates the screen interlacing even and odd scan lines to reduce tearing artifacts (if the decompressor supports this mode).

codecFlagCatchUpDiff

This input flag notifies the codec that the currently displayed frame is being displayed late in an attempt to "catch up" to the current frame, which only happens with compression formats that support frame differencing.

asyncCompletionProc

Points to a completion function structure. The compressor calls your completion function when an asynchronous decompression operation is complete. You can cause the decompression to be performed asynchronously by specifying a completion function. See *Inside Macintosh: QuickTime* for more information about completion functions.

If you specify asynchronous operation, you must not read the decompressed image until the decompressor indicates that the operation is complete by calling your completion function. Set `asyncCompletionProc` to `nil` to specify synchronous decompression. If you set `asyncCompletionProc` to –1, the operation is performed asynchronously but the decompressor does not call your completion function.

`frameTime` Points to a structure that contains the frame's time information, including the time at which the frame should be displayed, its duration, and the movie's playback rate. This parameter can be nil, in which case the decompression operation will happen immediately.

DISCUSSION

This function, introduced with QuickTime 2.0, accepts the same parameters as the `DecompressSequenceFrame` function, with the addition of the `frameTime` and `dataSize` parameters. The `frameTime` parameter points to an `ICMFrameTime` structure, which contains the frame's time information. The `ICMFrameTime` structure is described in Chapter 4, "Image Compressor Components."

SPECIAL CONSIDERATIONS

If the current decompressor component does not support scheduled asynchronous decompression, the Image Compression Manager returns an error code of `codecCantWhenErr`. In this case, the application will need to reissue the request with the `frameTime` parameter set to nil. If the decompressor cannot service your request at a particular time (for example, if its queue is full), the Image Compression Manager returns an error code of `codecCantQueueErr`. The best way to determine whether a decompressor component supports this function is to call the function and test the result code. A decompressor's ability to honor the request may change based on screen depth, clipping settings, and so on.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	Could not find the specified decompressor
`codecSpoolErr`	–8966	Error loading or unloading data
`codecCantWhenErr`	–8974	Decompressor can't honor this request
`codecCantQueueErr`	–8975	Decompressor can't queue this frame

DecompressSequenceFrameS

Queues a frame for decompression and specifies the size of the compressed data. New applications should use "DecompressSequenceFrameWhen" (page 3-13).

```
pascal OSErr DecompressSequenceFrameS(
                    ImageSequence seqID,
                    Ptr data,
```

```
        long dataSize,
        CodecFlags inFlags,
        CodecFlags *outFlags,
        ICMCompletionProcRecordPtr asyncCompletionProc);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.

`data` Points to the compressed image data. This pointer must contain a 32-bit clean address. If you use a dereferenced, locked handle, you must call the Memory Manager's `StripAddress` function before you use that pointer with this parameter.

`dataSize` Specifies the size of the `data` buffer.

`inFlags` Contains flags providing further control information. See *Inside Macintosh: QuickTime* for information about `CodecFlags` fields. The following flags are valid for this function:

`codecFlagNoScreenUpdate`
Controls whether the decompressor updates the screen image. If you set this flag to 1, the decompressor does not write the current frame to the screen, but does write the frame to its offscreen image buffer (if one was allocated). If you set this flag to 0, the decompressor writes the frame to the screen.

`codecFlagDontOffscreen`
Controls whether the decompressor uses the offscreen buffer during sequence decompression. This flag is only used with sequences that have been temporally compressed. If this flag is set to 1, the decompressor does not use the offscreen buffer during decompression. Instead, the decompressor returns an error. This allows your application to refill the offscreen buffer. If this flag is set to 0, the decompressor uses the offscreen buffer if appropriate.

`codecFlagOnlyScreenUpdate`
Controls whether the decompressor decompresses the current frame. If you set this

flag to 1, the decompressor writes the contents of its offscreen image buffer to the screen, but does decompress the current frame. If you set this flag to 0, the decompressor decompresses the current frame and writes it to the screen. You can set this flag to 1 only if you have allocated an offscreen image buffer for use by the decompressor.

`outFlags` Contains status flags. The decompressor updates these flags at the end of the decompression operation. See *Inside Macintosh: QuickTime* for information about `CodecFlags` constants. The following flags may be set by this function:

`codecFlagUsedNewImageBuffer`
Indicates to your application that the decompressor used the offscreen image buffer for the first time when it processed this frame. If this flag is set to 1, the decompressor used the image buffer for this frame and this is the first time the decompressor used the image buffer in this sequence.

`codecFlagUsedImageBuffer`
Indicates whether the decompressor used the offscreen image buffer. If the decompressor used the image buffer during the decompress operation, it sets this flag to 1. Otherwise, it sets this flag to 0.

`codecFlagDontUseNewImageBuffer`
This input flag forces an error to be returned when a new image buffer would have to be allocated instead of allocating the new buffer.

`codecFlagInterlaceUpdate`
This input flag updates the screen interlacing even and odd scan lines to reduce tearing artifacts (if the decompressor supports this mode).

`codecFlagCatchUpDiff`
This input flag notifies the codec that the currently displayed frame is being displayed

late in an attempt to "catch up" to the current frame, which only happens with compression formats that support frame differencing.

`asyncCompletionProc`

Points to a completion function structure. The compressor calls your completion function when an asynchronous decompression operation is complete. You can cause the decompression to be performed asynchronously by specifying a completion function. See *Inside Macintosh: QuickTime* for more information about completion functions.

If you specify asynchronous operation, you must not read the decompressed image until the decompressor indicates that the operation is complete by calling your completion function. Set `asyncCompletionProc` to `nil` to specify synchronous decompression. If you set `asyncCompletionProc` to –1, the operation is performed asynchronously but the decompressor does not call your completion function.

DISCUSSION

This function, introduced in QuickTime 1.6.1, accepts the same parameters as the `DecompressSequenceFrame` function, with the addition of the `dataSize` parameter.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	Could not find the specified decompressor
`codecSpoolErr`	–8966	Error loading or unloading data

CDSequenceFlush

Stops a decompression sequence, aborting processing of any queued frames.

```
pascal OSErr CDSequenceFlush(ImageSequence seqID);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.

DISCUSSION

This function, introduced with QuickTime 2.0, is used to tell a decompressor component to stop processing of any queued scheduled asynchronous decompression. This is useful when several frames have been queued for decompression in the future and the application needs to suspend playback of the sequence.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

SetDSequenceTimeCode

Sets the timecode value for the frame that is about to be decompressed.

```
pascal OSErr SetDSequenceTimeCode (
                    ImageSequence seqID,
                    const TimeCodeDef *timeCodeFormat,
                    const TimeCodeTime *timeCodeTime);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.

`timeCodeFormat`
Contains a pointer to a timecode definition structure. You provide the appropriate timecode definition information for the next frame to be decompressed.

`timeCodeTime` Contains a pointer to a timecode record structure. You provide the appropriate time value for the next frame in the current sequence.

DISCUSSION

QuickTime's video media handler uses this function to set the timecode information for a movie. When a movie that contains timecode information starts playing, the media handler calls this function as it processes the movie's first frame.

Note that the Image Compression Manager passes the timecode information straight through to the image decompressor component. That is, the Image Compression Manager does not make a copy of any of this timecode information. As a result, you must make sure that the data referred to by the `timeCodeFormat` and `timeCodeTime` parameters is valid until the next decompression operation completes.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	Could not find the specified decompressor

CDSequenceEquivalentImageDescription

Reports whether two image descriptions are the same.

```
pascal OSErr CDSequenceEquivalentImageDescription (
                    ImageSequence seqID,
                    ImageDescriptionHandle newDesc,
                    Boolean *equivalent);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.

`newDesc` Contains a handle to the image description structure that describes the compressed image.

`equivalent` Contains a pointer to a Boolean value. If the `ImageDescriptionHandle` provided in the `newDesc` parameter is equivalent to the image description currently in use by the image sequence, this value is set to `true`. If the

ImageDescriptionHandle is not equivilent, and therefore a new image sequence must be created to display an image using the new image description, this value is set to false.

DISCUSSION

The CDSequenceEquivalentImageDescription function allows an application to ask whether two image descriptions are the same. If they are, the decompressor does not have to create a new image decompression sequence to display those images.

SPECIAL CONSIDERATIONS

The Image Compression Manager can only implement part of this function by itself. There are some fields in the image description that it knows are irrelevant to the decompressor. If the Image Compression Manager determines that there are differences in fields that may be significant to the codec, it calls the ImageCodecIsImageDescriptionEquivalent function (page 4-25) to ask the codec.

CDSequenceNewMemory

Requests codec-allocated memory.

```
pascal OSErr CDSequenceNewMemory (
                    ImageSequence seqID,
                    Ptr *data,
                    Size dataSize,
                    long dataUse,
                    ICMMemoryDisposedUPP memoryGoneProc,
                    void *refCon);
```

seqID	Contains the unique sequence identifier that was returned by the DecompressSequenceBegin function.
data	Returns a pointer to the allocated memory.
dataSize	Specifies the requested size of the data buffer.

`dataUse` A code that indicates how the memory is to be used. For example, the memory may be used to store compressed data before it's displayed, mask plane data, or compressed data.

If there is no benefit to storing a particular kind of data in codec memory, the codec should deny the request for the memory allocation. The defined values are for data use are:

0x0001	Memory will be used for holding compressed image data.
0x0002	Memory will be used for an offscreen image buffer.

`memoryGoneProc`

A pointer to a function that will be called before disposing of the memory allocated by a codec. Your callback function must be in the following form:

```
pascal void (*ICMMemoryDisposedProcPtr)
             (Ptr memoryBlock, void *refcon);
```

`refCon` Contains a reference constant value to be passed to your `memoryGoneProc` function.

DISCUSSION

Because many newer hardware decompresson boards contain dedicated on-board memory, significant performance gains can be realized if this memory is used to store data before it is decompressed.

The decompressor can, at any time, dispose of all memory it has allocated. When memory is allocated, an `ICMMemoryDisposedProc` callback function must be provided. The decompressor calls this routine before disposing of the memory.

A callback procedure is required because memory on the hardware decompresson board may be limited. If the decompressor cannot deallocate memory as required, it is possible that an idle decompressor instance may be holding a large amount of memory, denying those resources to the currently active decompressor instance.

The decompressor memory must never be disposed at interrupt time. When the procedure is called, the memory is still available. This allows any pending reads into the block to be canceled before the block is disposed. The

decompressor disposing the memory must ensure that it is not disposing a block that it is currently using (that is, the memory that contains the currently decompressing frame).

To dispose of the memory, use the `CDSequenceDisposeMemory` function.

CDSequenceDisposeMemory

Disposes of memory allocated by the codec.

```
pascal OSErr CDSequenceDisposeMemory (
                    ImageSequence seqID,
                    Ptr data);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.

`data` Points to the previously allocated memory block.

DISCUSSION

You call this function to release memory allocated by the `CDSequenceNewMemory` function.

SPECIAL CONSIDERATIONS

Do not call the `CDSequenceDisposeMemory` function at interrupt time.

CDSequenceInvalidate

Notifies the Image Compression Manager that the destination port for the given image decompression sequence has been invalidated.

```
pascal OSErr CDSequenceInvalidate(
                    ImageSequence seqID,
                    RgnHandle invalRgn);
```

`seqID`	Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.
`rgn`	A handle of the region specifiying the invalid portion of the image.

DISCUSSION

You call this function to force the Image Compression Manager to redraw the screen bits on the next decompression operation.

Working With Images

PtInDSequenceData

Tests to see if an image contains data at a a given point.

```
pascal OSErr PtInDSequenceData(
                    ImageSequence seqID,
                    void *data,
                    Size dataSize,
                    Point where,
                    Boolean *hit);
```

`seqID`	Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.
`data`	Pointer to compressed data in the format specified by the `desc` param.
`dataSize`	Size of the compressed data referred to by the `data` param.
`where`	A QuickDraw Point. 0,0 based at the top-left corner of the image.

`hit` A pointer to a field to receive the Boolean indicating whether or not the image contained data at the specified point. The Boolean will be set to `true` if the point specified by the `where` parameter is contained within the compressed image data specified by the `data` param.

DISCUSSION

The `PtInDSequenceData` function allows the application to perform hit testing on compressed data. The `hit` parameter will be set to true if the compressed data contains data at the point specified by the `where` parameter. The `hit` parameter will be set to false if the specified point falls within a blank portion of the image.

Working With Data Sources

CDSequenceNewDataSource

Creates a new data source.

```
pascal OSErr CDSequenceNewDataSource (
                    ImageSequence seqID,
                    ImageSequenceDataSource *sourceID,
                    OSType sourceType,
                    long sourceInputNumber,
                    Handle dataDescription,
                    void *transferProc,
                    void *refCon);
```

`seqID` The unique sequence identifier that was returned by the `DecompressSequenceBegin` function.

`sourceID` Returns the new data source identifier.

`sourceType` A four-character code describing how the input will be used. This code is usually derived from the information returned by the codec. For example, if a mask plane was passed, this field might contain ‘mask’.

`sourceInputNumber`
: More than one instance of a given source type may exist. The first occurrence should have a source input number of 1, the second a source input number of 2, and so on.

`dataDescription`
: A handle to a data structure describing the input data. For compressed image data, this is just an `ImageDescriptionHandle`.

`transferProc`
: A routine that allows the application to transform the type of the input data to the kind of data preferred by the codec. The client of the codec passes the source data in the form most convenient for it. If the codec needs the data in another form, it can negotiate with the client or directly with the Image Compression Manager to obtain the required data format.

`refCon`
: Contains a reference constant value to be passed to the transfer procedure.

DISCUSSION

This function returns a `sourceID` parameter which must be passed to all other functions that reference the source. All data sources are automatically disposed when the sequence they are associated with is disposed.

CDSequenceDisposeDataSource

Disposes of a data source.

```
pascal OSErr CDSequenceDisposeDataSource (
                    ImageSequenceDataSource sourceID);
```

`sourceID`
: The data source identifier that was returned by the `CDSequenceNewDataSource` function.

DISCUSSION

You use this function to dispose of a data source created by the `CDSequenceNewDataSource` function. All data sources are automatically disposed when the sequence they are associated with is disposed.

CDSequenceSetSourceData

Sets data in a new frame to a specific data source.

```
pascal OSErr CDSequenceSetSourceData (
                    ImageSequenceDataSource sourceID,
                    void *data,
                    long dataSize);
```

`sourceID` Contains the source identifier of the data source.

`data` Points to the data. This pointer must contain a 32-bit clean address. If you use a dereferenced, locked handle, you must call the Memory Manager's `StripAddress` function before you use that pointer with this parameter.

`dataSize` Specifies the size of the `data` buffer.

DISCUSSION

The `CDSequenceSetSourceData` function is called to set data in a new frame to a specific source. For example, as a new frame of compressed data arrives at a source, `CDSequenceSetSourceData` will be called.

CDSequenceChangedSourceData

Notifies the compressor that the image source data has changed.

```
pascal OSErr CDSequenceChangedSourceData (
                    ImageSequenceDataSource sourceID);
```

`sourceID` Contains the source identifier of the data source.

DISCUSSION

Use the new `CDSequenceSetChangedSourceData` function to indicate that the image has changed but the data pointer to that image has not changed. For example, if the data pointer points to the base address of a PixMap. The image in the PixMap can change, but the data pointer remains constant.

Working With Image Description Records

AddImageDescriptionExtension

Adds an extension to an `ImageDescriptionHandle`.

```
pascal OSErr AddImageDescriptionExtension(
                    ImageDescriptionHandle desc,
                    Handle extension,
                    long idType);
```

`desc`	The handle of the ImageDescription to add the extension to.
`extension`	The handle containing the extension data.
`idType`	A four-byte signature indentifying the type of data being added to the ImageDescription.

DISCUSSION

This function allows the application to add custom data to an ImageDescriptionHandle. This data could be specific to the compressor component referenced by the image description.

SPECIAL CONSIDERATIONS

The Image Compression Manager makes a copy of the data referred to by the `extension` parameter. Thus, your application should dispose its copy of the data when it is no longer needed.

GetNextImageDescriptionExtensionType

Adds an extension to an `ImageDescriptionHandle`.

```
pascal OSErr GetNextImageDescriptionExtensionType(
                    ImageDescriptionHandle desc,
                    long *idType);
```

`desc` The `ImageDescriptionHandle`.

`idType` A pointer to a field that, on entry, contains the starting point for the search. On return, will contain the next extension type found in the `ImageDescriptionHandle`.

DISCUSSION

This function allows the application to search for all types of extensions in an `ImageDescriptionHandle`. The `idType` field should be set to 0 to start the search. When no more extension types can be found, this field will be set to 0.

CountImageDescriptionExtensionType

Counts the number of extensions of a given type in an `ImageDescriptionHandle`.

```
pascal OSErr CountImageDescriptionExtensionType(
                    ImageDescriptionHandle desc,
                    long idType,
                    long *count);
```

`desc` The `ImageDescriptionHandle`.

`idType` A four-byte signature indentifying the type of extension data.

`count` A pointer to a field to receive the number of extensions of the specified type.

DISCUSSION

This function, when used with the "GetNextImageDescriptionExtensionType" call, allows the application to determine the total set of extensions present in the ImageDescriptionHandle.

GetImageDescriptionExtension

Returns a new handle with the data from a specified image description extension.

```
pascal OSErr GetImageDescriptionExtension(
                    ImageDescriptionHandle desc,
                    Handle *extension,
                    long idType,
                    long index);
```

desc	The ImageDescriptionHandle.
extension	A pointer to a field to receive a new handle with the extension data.
idType	The type of extension to receive.
index	The index (from 1 to the count as returned by "CountImageDescriptionExtensionType") of the extension to receive.

DISCUSSION

This function allows the application to get a copy of a specified image description extension.

SPECIAL CONSIDERATIONS

The Image Compression Manager allocates a new handle and passes it back in the extension parameter. Your application should dispose of the handle when it is no longer needed.

RemoveImageDescriptionExtension

Removes a specified extension from an `ImageDescriptionHandle`.

```
pascal OSErr RemoveImageDescriptionExtension(
                    ImageDescriptionHandle desc,
                    long idType,
                    long index);
```

`desc` The `ImageDescriptionHandle`.

`idType` The type of extension to receive.

`index` The index (from 1 to the count as returned by "`CountImageDescriptionExtensionType`") of the extension to receive.

DISCUSSION

This function allows the application to remove a specified extension from an `ImageDescriptionHandle`. Note that any extensions that are present in the `ImageDescriptionHandle` after the deleted extension will have their index numbers renumbered.

Changing Sequence Compression Parameters

SetCSequencePreferredPacketSize

Sets the preferred packet size for a sequence.

```
pascal OSErr SetCSequencePreferredPacketSize (
                    ImageSequence seqID,
                    long preferredPacketSizeInBytes);
```

`seqID` The sequence identifier.

`preferredPacketSizeInBytes`
The preferred packet size in bytes.

DISCUSSION

This function was added in QuickTime 2.5 to support video conferencing applications.

Controlling Hardware Scaling

QuickTime 1.6.1 added three functions that allow applications to zoom a monitor (`GDHasScale`, `GDGetScale`, and `GDSetScale`). These three functions are considered low-level calls (comparable to `SetEntries`) that you should use only when playing back QuickTime movies in a controlled environment with no user interaction. Also, because this capability is not present on all machines, applications should not depend on its availability.

These new functions provide a standard way for you to access the resizing abilities of a user's monitor for playback. Effectively, this allows you to have full screen Cinepak playback on low-end Macintosh computers.

Hardware 200 percent resize is currently available only on the Macintosh LC II, IIvx, IIvi, Performa 400, Performa 600, and Color Classic in 16-bit display mode on the 12-inch (512 x 384) monitors.

GDHasScale

Returns the closet possible scaling that a particular screen device can be set to in a given pixel depth.

```
pascal OSErr GDHasScale (
                    GDHandle gdh,
                    short depth,
                    Fixed *scale);
```

`gdh` Contains a handle to a screen graphics device.

`depth` Specifies the pixel depth of the screen device.

`scale` Points to a fixed point scale value. On input, this field should be set to the desired scale value. On output, this field will contain the closest scale available for the given depth. A scale of 0x10000 indicates normal size, 0x20000 indicates double size, and so on.

DISCUSSION

The `GDHasScale` function returns scaling information for a particular GDevice for a requested depth. This function allows you to query a GDevice without actually changing it. For example, if you specify 0x20000 but the GDevice does not support it, `GDHasScale` returns with noErr and a scale of 0x10000. Because this function checks for a supported depth, your requested depth must be supported by the GDevice. `GDHasScale` references the video driver through the graphics device structure.

RESULT CODES

`CDepthErr`	–157	The requested depth is not supported
`cDevErr`	–155	Not a screen device
`controlErr`	–17	Video driver cannot respond to this call

GDGetScale

Returns the current scale of the given screen graphics device.

```
pascal OSErr GDGetScale (
                    GDHandle gdh,
                    Fixed *scale,
                    short *flags);
```

`gdh` Contains a handle to a screen graphics device.

`scale` Points to a fixed point field to hold the scale result.

`flags` Points to a short integer. It returns the status parameter flags for the video driver. For now, 0 is always returned in this field.

RESULT CODES

cDevErr	–155	Not a screen device
controlErr	–17	Video driver cannot respond to this call

GDSetScale

Sets a screen graphics device to a new scale.

```
pascal OSErr GDSetScale (
                    GDHandle gdh,
                    Fixed scale,
                    short flags);
```

gdh	Contains a handle to a screen graphics device.
scale	A fixed point scale value.
flags	Points to a short integer. It returns the status parameter flags for the video driver. For now, 0 is always returned in this field.

RESULT CODES

cDevErr	–155	Not a screen device
controlErr	–17	Video driver cannot respond to this call

Working With Video Fields

QuickTime 2.5 introduces three functions for working with compressed fields of video data. The `ImageFieldSequenceBegin` function initiates an image field sequence operation; the `ImageFieldSequenceExtractCombine` function performs the desired operations; and the `ImageFieldSequenceEnd` function terminates the operation.

ImageFieldSequenceBegin

Initiates an image field sequence operation and specifies the input and output data format.

```
pascal OSErr ImageFieldSequenceBegin (
                    ImageFieldSequence *ifs,
                    ImageDescriptionHandle desc1,
                    ImageDescriptionHandle desc2,
                    ImageDescriptionHandle descOut);
```

`ifs` On return, contains the unique sequence identifier assigned to the sequence.

`desc1` An image description structure describing the format and characteristics of the data to be passed to the `ImageFieldSequenceExtractCombine` function through the `data1` parameter.

`desc2` An image description structure describing the format and characteristics of the data to be passed to the `ImageFieldSequenceExtractCombine` function through the `data2` parameter. Set to `nil` if the requested operation uses only one input frame.

`descOut` Specifies the desired format of the resulting frames. Typically this is the same format specified by the `desc1` and `desc2` parameters.

DISCUSSION

You use the `ImageFieldSequenceBegin` function to set up an image field sequence operation and specify the input and output data format.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available

ImageFieldSequenceExtractCombine

Performs field operations on video data.

```
pascal OSErr ImageFieldSequenceExtractCombine (
                    ImageFieldSequence ifs,
                    long fieldFlags,
                    void *data1,
                    long dataSize1,
                    void *data2,
                    long dataSize2,
                    void *outputData,
                    long *outDataSize);
```

`ifs` The unique sequence identifier that was returned by the `ImageFieldSequenceBegin` function.

`fieldFlags` Flags specifying the operation to be performed. A correctly formed request will specify two input fields, mapping one to the odd output field and the other to the even output field. The following flags are defined:

`evenField1ToEvenFieldOut`
Maps the even field specified by the `data1` parameter to the even output field.

`evenField1ToOddFieldOut`
Maps the even field specified by the `data1` parameter to the odd output field.

`oddField1ToEvenFieldOut`
Maps the odd field specified by the `data1` parameter to the even output field.

oddField1ToOddFieldOut
: Maps the odd field specified by the `data1` parameter to the odd output field.

evenField2ToEvenFieldOut
: Maps the even field specified by the `data2` parameter to the even output field.

evenField2ToOddFieldOut
: Maps the even field specified by the `data2` parameter to the odd output field.

oddField2ToEvenFieldOut
: Maps the odd field specified by the `data2` parameter to the even output field.

oddField2ToOddFieldOut
: Maps the odd field specified by the `data2` parameter to the odd output field.

`data1` A pointer to a buffer containing the data of input field one.

`dataSize1` Specifies the size of the `data1` buffer.

`data2` A pointer to a buffer containing the data of input field two. Set to `nil` if the requested operation uses only one input frame.

`dataSize2` Specifies the size of the `data2` buffer. Set to 0 if the requested operation uses only one input frame.

`outputData` A pointer to a buffer to receive the resulting frame. Use the `GetMaxCompressionSize` function to determine the amount of memory to allocate for this buffer.

`outDataSize` On output this parameter returns the actual size of the data.

DISCUSSION

This function was introduced in QuickTime 2.5 and provides a method for working directly with fields of interlaced video. You can use the `ImageFieldSequenceExtractCombine` function to change the field dominance of an image by reversing the two fields, or to create or remove the effects of the 3:2 pulldown commonly performed when transferring film to NTSC videotape.

Because this function operates directly on the compressed video data, it is faster than working with decompressed images. It also has the added benefit of eliminating any image quality degradation that might result from lossy codecs.

The `ImageFieldSequenceExtractCombine` function accepts one or two compressed images as input and creates a single compressed image on output. You specify the operation to be performed using the `fieldFlags` parameter. The function returns the `codecUnimpErr` result code if there is no codec present in the system that can perform the requested operation.

The Apple Component Video (YUV) and Motion JPEG codecs currently support this function. See Chapter 4, "Image Compressor Components," for information on incorporating support for this function in your codec.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`codecUnimpErr`	–8962	Feature not implemented by this compressor

ImageFieldSequenceEnd

Ends an image field sequence operation.

```
pascal OSErr ImageFieldSequenceEnd (ImageFieldSequence ifs);
```

`ifs` The unique sequence identifier that was returned by the `ImageFieldSequenceBegin` function.

DISCUSSION

You must call this function to terminate an image field sequence operation.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified

Image Transcoding Functions

ImageTranscodeSequenceBegin

Initiates an image transcoder sequence operation.

```
pascal OSErr ImageTranscodeSequenceBegin (
                    ImageTranscodeSequence *its,
                    ImageDescriptionHandle srcDesc,
                    OSType destType,
                    ImageDescriptionHandle *dstDesc
                    void *data,
                    long dataSize);
```

its	The image transcoder sequence identifier. If the operation fails, the value pointed to by its is set to nil.
srcDesc	The image description for the source compressed image data.
destType	The desired compression format into which to transcode the source data.
dstDesc	Returns an image description for the data which will be generated by the image transcoding sequence.
data	Pointer to first frame of compressed data to transcode. Set to nil of not available.
dataSize	Size of the compressed data, in bytes. Set to zero if no data is provided.

DISCUSSION

This function begins an image transcoder sequence operation and returns the sequence identifier in the its parameter. The caller is responsible for disposing

of the image description that is returned in the `dstDesc` parameter. If no transcoder is available to perform the requested transcoding operation, a `handlerNotFound` error is returned.

ImageTranscodeFrame

Transcodes a frame of image data.

```
pascal OSErr ImageTranscodeFrame (
                    ImageTranscodeSequence its,
                    void *srcData,
                    long srcDataSize,
                    void **dstData,
                    long *dstDataSize);
```

`its`	Specifies the image transcoder sequence to use to perform the transcoding operation.
`srcData`	Contains a pointer to the source data to transcode.
`srcDataSize`	Indicates the size of the compressed source image data in bytes.
`dstData`	Returns a pointer to the transcoded image data.
`dstDataSize`	Returns the size of the transcoded image data.

DISCUSSION

After creating the image transcoder sequence using `ImageTranscodeSequenceBegin`, you use the `ImageTranscodeFrame` function to transcode a frame of image data. The caller is responsible for disposing of the transcoded data using the `ImageTranscodeDisposeFrameData` function.

ImageTranscodeDisposeFrameData

Disposes transcoded image data.

```
pascal OSErr ImageTranscodeDisposeFrameData(
                    ImageTranscodeSequence its,
                    void *dstData)
```

`its` Specifies the image transcoder sequence that was used to generate the transcoded data.

`dstData` Contains a pointer to the transcoded image data generated by the `ImageTranscodeFrame` function.

DISCUSSION

When the transcoded image data returned by `ImageTranscodeFrame` is no longer needed, use the `ImageTranscodeDisposeFrameData` function to dispose of the data. Only the image transcoder that generated the data can properly dispose of it.

ImageTranscodeSequenceEnd

Ends an image transcoder sequence operation.

The only parameter to `ImageTranscodeSequenceEnd` is the identifier of the image transcoder sequence to dispose. It is safe to pass a value of 0 to this routine.

```
pascal OSErr ImageTranscodeSequenceEnd (ImageTranscodeSequence its)
```

`its` The identifier of the image transcoder sequence to dispose. It is safe to pass a value of 0 in this parameter.

DISCUSSION

You must call this function to terminate an image transcoder sequence operation and dispose of the sequence.

GetGraphicsImporterForFile

Locates and opens a graphics importer component that can be used to draw the specified file.

```
pascal OSErr GetGraphicsImporterForFile(
                    const FSSpec *theFile,
                    ComponentInstance *gi);
```

`theFile` Specifies the file to be drawn using a graphics importer component.

`gi` A pointer to a `ComponentInstance` in which the best graphics importer for working with the specified file will be returned. If no graphics importer can be found, the `ComponentInstance` will be set to nil.

DISCUSSION

`GetGraphicsImporterForFile` first tries to locate a graphics importer component for the specified file based on the Macintosh file type of the file. If it is unable to locate a graphics importer component based on the Macintosh file type, and the file type is 'TEXT', `GetGraphicsImporterForFile` will try to locate a graphics importer component for the specified file based on the file name extension of the file.

If it is unable to locate a graphics importer for the file, the returned `ComponentInstance` is set to nil. If it is able to locate a graphics importer for the file, the returned graphics importer `ComponentInstance` will have already been set up to draw the specified file in the current port.

The caller of `GetGraphicsImporterForFile` is responsible for closing the returned `ComponentInstance` using `CloseComponent`.

GetGraphicsImporterForDataRef

Locates and opens a graphics importer component that can be used to draw the specified data reference.

```
pascal OSErr GetGraphicsImporterForDataRef(
                    Handle dataRef,
                    OSType dataRefType,
                    ComponentInstance *gi);
```

`dataRef` Specifies the data reference to be drawn using a graphics importer component.

`dataRefType` The type of the data reference specified by the `dataRef` parameter. For alias-based data references, the `dataRef` handle contains an AliasRecord, and `dataRefType` is equal to `rAliasType`.

`gi` A pointer to a `ComponentInstance` in which the best graphics importer for working with the specified data reference will be returned. If no graphics importer can be found, the `ComponentInstance` will be set to nil.

DISCUSSION

`GetGraphicsImporterForDataRef` tries to locate a graphics importer component for the specified data reference based on the file name extension of the file. The file name extension is retrieved from the data reference by use of the `DataHGetFileName` call to the data handler associated with the data reference.

If it is unable to locate a graphics importer for the file, the returned `ComponentInstance` is set to nil. If it is able to locate a graphics importer for the data reference, the returned graphics importer `ComponentInstance` will have already been set up to draw the specified data reference in the current port.

The caller of `GetGraphicsImporterForDataRef` is responsible for closing the returned `ComponentInstance` using `CloseComponent`.

CHAPTER 4

Image Compressor Components

Contents

This chapter discusses new features and changes to image compressor components as documented in Chapter 4 of *Inside Macintosh: QuickTime Components*.

New Features of Image Compressor Components

Asynchronous Decompression

In QuickTime 2.0 the Image Compression Manager was enhanced to support scheduled asynchronous decompression operations. By calling the Image Compression Manager function `DecompressSequenceFrameWhen` (page 3-13), applications can schedule decompression requests in advance. This allows decompressor components that support this functionality to provide reliable playback performance under a wider range of conditions.

The Apple Cinepak, Video, Animation, Component Video, and Graphics decompressors provided in QuickTime versions 2.0 and later support scheduled asynchronous decompression to 8-, 16-, and 32-bit destinations (the Cinepak decompressor also supports 4-bit grayscale destinations). QuickTime 2.5 adds asynchronous decompression support to the JPEG and None decompressor components on PowerPC systems (with the QuickTime PowerPlug extension installed).

If you want to support this functionality, you must modify your decompressor component in the following ways:

- Report your component's new capabilities in its compressor capability structure by setting the `codecCanAsyncWhen` and `codecCanAsync` flags.
- Modify your component's `ImageCodecBandDecompress` function (page 4-23) to accept scheduled asynchronous decompression requests and process them correctly
- Implement the new `ImageCodecFlush` function (page 4-24); this function allows the Image Compression Manager to instruct you to empty your input queue
- Optionally, implement logic to manage the shielding of the cursor during decompression operations

All of these changes are discussed in detail in the reference section.

Hardware Cursors

The Image Compression Manager supports hardware cursors introduced in PCI-based Macintosh computers, which eliminates cursor flicker. For all software codecs this support requires no changes.

For codecs that manage the cursor themselves, QuickTime 2.1 provides a new flag, `codecCompletionDontUnshield`, for use when calling the `ICMDecompressComplete` (page 4-36) function. Use this flag to prevent the Image Compression Manager from unshielding the cursor when `ICMDecompressComplete` is called.

Timecode Support

As discussed in Chapter 3, "Image Compression Manager," timecode support was added to the Image Compression Manager in QuickTime 2.0. QuickTime 2.0 and above has continued to be enhanced to provide timecode information to decompressor components when movies are played. This feature is provided for hardware systems that may want to use timecode information.

To support timecode in your image compressor component, your codec must support the `CDCodecSetTimeCode` function (page 4-24), which allows the Image Compression Manager to set the timecode value for the next frame to be decompressed. For more information about timecode tracks and the timecode media handler, see Chapter 1, "Movie Toolbox."

Working With Video Fields

The functionality of the `ImageFieldSequenceExtractCombine` function described in Chapter 3, "Image Compression Manager," is performed by individual image codecs. This is because the way in which fields are stored is different for every compression format that supports text. A new codec component function, `ImageCodecExtractAndCombineFields` (page 4-19), has been defined for this purpose. Apple encourages developers of codecs to incorporate this function, if their compressed data format is capable of separately storing both fields of a video frame.

Accelerated Video Support

QuickTime 2.5 contains new support for developers of codecs to accelerate certain image decompression operations. These features will most likely be used by developers of video hardware boards that provide special acceleration features, such as arbitrary scaling or color space conversion.

Prior to QuickTime 2.5, if a codec could not decompress an image directly to the screen, the ICM would prepare an offscreen buffer for the codec, then use the None codec to transfer the image from the offscreen buffer to the screen. With 2.5, if a codec cannot decompress directly to the screen, it has the option of specifying that it can decompress to one or more types of non-RGB pixel spaces, specified as an OSType (e.g., 'yuvs'). The ICM will then attempt to find a decompressor component of that type (a transfer codec) that can transfer the image to the screen. Since the ICM does not define non-RGB pixel types, the transfer codec must support additional calls to set up the offscreen. If a transfer codec cannot be found that supports the specified non-RGB pixel types, the ICM will use the None codec with an RGB offscreen buffer.

The real speed benefit comes from the fact that since the transfer codec defines the offscreen buffer, it can place the buffer in on-board memory, or even point to an overlay plane so that the offscreen really is on screen. In this case, the additional step of transferring the bits from offscreen memory on to the screen is avoided.

For an image decompressor component to indicate that it can decompress to non-RGB pixel types, it should, in the `ImageCodecPreDecompress` call, fill in the `wantedDestinationPixelTypes` field with a handle to a zero-terminated list of pixel types that it can decompress to. The ICM immediately makes a copy of the handle. Cinepak, for example, returns a 12-byte handle containing `yuvs`, `yuvu`, and $00000000. Since `ImageCodecPreDecompress` can be called often, it is suggested that codecs allocate this handle when their component is opened and simply fill in the `wantedDestinationPixelTypes` field with this handle during `ImageCodecPreDecompress`. Components that use this method should be sure to dispose the handle at close.

Apple's Cinepak decompressor supports decompressing to 'yuvs' and 'yuvu' pixel types. 'yuvs' is a YUV format with u and v components signed (center point at $00), while 'yuvu' has the u and v component centered at $80. The YUV format used by QuickTime is documented in **<<••••movie file format docs???•••>>**.

As an example, suppose XYZ Co. had a video board that had a YUV overlay plane capable of doing arbitrary scaling. The overlay plane takes data in the

same format as Cinepak's yuvs format. In this case, XYZ would make a component of type 'imdc' and subtype 'yuvs'.

The `CDPreDecompress` call would set the `codecCanScale`, `codecHasVolatileBuffe`r, and `codecImageBufferIsOnScreen` bits in the capabilities->flags field. The `codecImageBufferIsOnScreen` bit is necessary to inform the ICM that the codec is a direct screen transfer codec. A direct screen transfer codec is one that sets up an offscreen buffer that is actually onscreen (such as an overlay plane). Not setting this bit correctly can cause unpredictable results.

The real work of the codec takes place in the `CDCodecNewImageBufferMemory` call. This is where the codec is instructed to prepare the non-RGB pixel buffer. The codec must fill in the `baseAddr` and `rowBytes` fields of the `dstPixMa`p structure in the `CodecDecompressParams`. The ICM then uses passes these values to the original codec (e.g., Cinepak) to decompress into.

The codec must also implement `CDDisposeMemory` to balance `CDCodecNewImageBufferMemory`.

Since Cinepak then decompresses into the card's overlay plane, `CDBandDecompress` needs to do nothing aside from calling `ICMDecompressComplete`.

```
pascal ComponentResult
CDPreDecompress(Handle storage,
        CodecDecompressParams *p)
{
    CodecCapabilities*capabilities = p->capabilities;

    // only allow 16 bpp source
    if ((**p->imageDescription).depth != 16)
        return codecConditionErr;

    /* we only support 16 bits per pixel dest */
    if (p->dstPixMap.pixelSize != 16)
        return codecConditionErr;

    capabilities->wantedPixelSize = p->dstPixMap.pixelSize;

    capabilities->bandInc = capabilities->bandMin =
                (*p->imageDescription)->height;
```

```
    capabilities->extendWidth = 0;
    capabilities->extendHeight = 0;

    capabilities->flags =
            codecCanScale | codecImageBufferIsOnScreen |
            codecHasVolatileBuffer;

    return noErr;
}

pascal ComponentResult

CDBandDecompress(Handle storage,
        CodecDecompressParams *p)
{
    ICMDecompressComplete(p->sequenceID, noErr,
                codecCompletionSource | codecCompletionDest,
                &p->completionProcRecord);

    return noErr;
}

pascal ComponentResult
CDCodecNewImageBufferMemory(Handle storage,
        CodecDecompressParams *p, long flags,
        ICMMemoryDisposedUPP memoryGoneProc,
        void *refCon)
{
    OSErr err = noErr;
    long offsetH, offsetV;
    Ptr baseAddr;
    long rowBytes;

    // call predecompress to check to make sure we can handle
    // this destination
    err = CDPreDecompress(storage, p);
    if (err) goto bail;

    // set video board registers with the scale
    XYZVideoSetScale(p->matrix);
```

```
    // calculate a base address to write to
    offsetH = (p->dstRect.left - p->dstPixMap.bounds.left);
    offsetV = (p->dstRect.top - p->dstPixMap.bounds.top);
    XYZVideoGetBaseAddress(p->dstPixMap, offsetH, offsetV,
                &baseAddr, &rowBytes);

    p->dstPixMap.baseAddr = baseAddr;
    p->dstPixMap.rowBytes = rowBytes;
    p->capabilities->flags = codecImageBufferIsOnScreen;

bail:
    return err;
}

pascal ComponentResult
CDDisposeMemory(Handle storage, Ptr  data)
{
    return noErr;
}
```

Some video hardware boards that use an overlay plane require that the image area on screen be flooded with a particular RGB value or alpha-channel in order to have the overlay buffer "show through" at that location. Codecs that require this support should set the `screenFloodMethod` and `screenFloodValue` fields of the `CodecDecompressParams` record during `ImageCodecPreDecompress`. The ICM will then manage the flooding of the screen buffer. This method is more reliable than having the codec attempt to flood the screen itself, and will ensure compatibility with future versions of QuickTime.

Packetization Information

QuickTime 2.5 includes new functions to support packetizing compressed data streams, primarily for video conferencing applications. These functions are discussed in Chapter 3, "Image Compression Manager," and other chapters of this book. In addition, a new field (`preferredPacketSizeInBytes`) has been added to the compression parameters structure (page 4-14). Codec developers need only use this field.

Packet information is appended, word-aligned, to the end of video data. It is a variable-length array of 4-byte integers, each representing the offset in bits of the end of a packet, followed by another integer containing the number of

packet hints, and finally a four-byte identifier indicating the type of appended data:

```
[boundary #1][boundary #2]...[boundary #N][N]['pkts']
```

Packets are given in bits, because some types of compressed image data (such as H.261) are cut up on bit-boundaries rather than byte-boundaries.

```
// given:   image data, length, and a packet number

// returns: a pointer to the start of the packet and a packet size, plus
information about leading and trailing bits

char* GetNextPacket(char* data, int len, int packet, long* packet_size,
    char* leading_bits, char* trailing_bits)
{
    long *lp, packets;
    lp = (long*) data;  // 'data' must be word-aligned
    lp += len/4 - 1;

    if (*lp != 'pkts')
        return nil;

    packets = *lp[-1];  // negative indexing is good for you
    if (packet >= packets)
        return nil;             // out of bounds

    lp -= packets;      // now 0-indexing into the packet array will work

    if (packet == 0)
    {
        *packet_size = (lp[0] + 7)/8;// count the bits
        *leading_bits = 0;
        *trailing_bits = lp[0] % 8;
        return data;            // in case of 0-length packet
    }
    else
    {
        *packet_size = ( lp[pktnum] - lp[pktnum-1] + 7) / 8;
        *leading_bits = lp[packet-1] % 8 ? 8 - lp[packet-1] % 8 : 0;
        *trailing_bits = lp[packet] % 8;
```

```
            return data + lp[packet-1] / 8;
        }
    }
```

Note that this can be used for further extensions with the addition of further append formats. The last two words will always be a number of words and an extension identifier.

Image Compressor Components Reference

Data Types

The Frame Time Structure

The `ICMFrameTime` structure is defined as follows:

```
struct ICMFrameTimeRecord {
    Int64Bit                value;          /* time to display frame */
    long                    scale;          /* time scale */
    void                    *base;          /* reference to time base */
    long                    duration;       /* display duration */
    Fixed                   rate;           /* movie's playback rate */
};
```

Field descriptions

`value`	Specifies the time at which the frame is to be displayed.
`scale`	Indicates the units for the frame's display time.
`base`	Refers to the time base.
`duration`	Specifies the duration for which the frame is to be displayed. This must be in the same units as specified by the `scale` field.
`rate`	Indicates the time base's effective rate.

The Decompression Data Source Structure

The `CDSequenceDataSource` structure contains a linked list of all data sources for a decompression sequence. Because each data source contains a link to the next data source, a codec can access all data sources from this field. The `CDSequenceDataSource` structure is defined as follows:

```
struct CDSequenceDataSource {
    long                        recordSize;
    void *                      next;
    ImageSequence               seqID;
    ImageSequenceDataSource     sourceID;
    OSType                      sourceType;
    long                        sourceInputNumber;
    void *                      dataPtr;
    Handle                      dataDescription;
    long                        changeSeed;
    ICMConvertDataFormatUPP     transferProc;
    void *                      transferRefcon;
    long                        dataSize;
};
typedef struct CDSequenceDataSource CDSequenceDataSource;
typedef CDSequenceDataSource *CDSequenceDataSourcePtr;
```

Field descriptions

`recordSize` Specifies the size of the record.

`next` Contains a pointer to the next source entry. If it is `nil`, there are no more entries.

`seqID` Specifies the image sequence that this source is associated with.

`sourceID` Specifies the source reference identifying this source.

`sourceType` A four-character code describing how the input will be used. This value is passed to this parameter by the `CDSequenceNewDataSource` function when the source is created.

`sourceInputNumber` This value is passed to this parameter by the `CDSequenceNewDataSource` function when the source is created.

`dataPtr` Contains a pointer to the actual source data.

`dataDescription`
: Contains a handle to a data structure describing the data format. This will often be a Image Description Handle.

`changeSeed`
: Contains an integer that is incremented each time the `dataPtr` field changes or that data that the `dataPtr` field points to changes. By remembering the value of this field and comparing to the value the next time the decompressor or compressor component is called, the component can determine if new data is present.

`transferProc`
: Reserved

`transferRefcon`
: Reserved

`datasize`
: Specifies the size of the `data` pointer to the `dataPtr` field.

The Compressor Capability Structure

Three new compressor capability flags have been added to the compressor capability structure. Your component sets these flags in the `flags` field of the `CodecCapabilities` structure:

```
enum {
    codecCanAsyncWhen           = 1L << 16,
    codecCanShieldCursor        = 1L << 17,
    codecCanManagePrevBuffer    = 1L << 18,
    codecHasVolatileBuffer      = 1L << 19
    codecImageBufferIsOnScreen  = 1L << 21,
    codecWantsDestinationPixels = 1L << 22
};
```

Flag descriptions

`codecCanAsyncWhen`
: Indicates whether your decompressor component supports scheduled asynchronous decompression. Set this flag to 1 if your component can support the scheduling functionality of the `CDBandDecompress` function. Note that you must also set the `codecCanAsync` flag to 1.

`codecCanShieldCursor`

Indicates whether your decompressor component will manage the shielding of the cursor during decompression. If your component can manage the cursor's display, set this flag to 1. Your component can use the Image Compression Manager's `ICMShieldSequenceCursor` function to shield the cursor. The cursor is automatically unsheilded when you call `ICMDecompressComplete`. This function is described later in this chapter in "Image Compression Manager Utility Functions."

Otherwise, set this flag to 0—the Image Compression Manager then manages the cursor for you.

It is highly recommended that you support this capability if your decompressor supports asynchronous operation or the cursor may remain shielded for unacceptably long periods of time.

`codecCanManagePrevBuffer`

Indicates that your compressor component is capable of allocating and managing the `prevPixMap` used in temporal compression. If this flag is set, then your compressor must determine when to update the `prevPixMap` during compression sequences. Codecs setting this flag should also set `codecCanCopyPrev`.

`codecHasVolatileBuffer`

Some hardware decompressors don't actually draw the decompressed pixels into the frame buffer as requested by QuickTime. Instead, they have a second frame buffer which floats or overlays above the main frame buffer. The image is decompressed into this secondary frame buffer instead. To the user, the effect is the same because the video hardware merges the two frame buffers together. When the window that contains the image is moved to another location in the same screen buffer, the Window Manager uses `CopyBits` to transfer the window's pixels from the old location to the new location. Unfortunately, because the Window Manager is unaware of the presence of the secondary frame buffer, it cannot move the image it is displaying.

By setting the `codecHasVolatileBuffer` flag to 1, the decompressor component informs QuickTime that it uses a secondary frame buffer. When the Window Manager moves a window, QuickTime forces a redraw of the contents of the window so that the secondary frame buffer can be repositioned and/or updated as necessary.

`codecImageBufferIsOnScreen`

By setting the `codecImageBufferIsOnScreen` flag to 1, the decompressor component informs QuickTime that it is a direct screen transfer codec. Codecs that use the `CDCodecNewImageBufferMemory` call to create an offscreen buffer that is really onscreen would set this flag. See the section "Accelerated Video Support" (page 4-5) for more information on this flag.

The Compression Parameters Structure

QuickTime 2.1 added three new fields to the compression parameters structure originally documented in *Inside Macintosh: QuickTime Components* to support data sources. QuickTime 2.5 added one additional field.

```
struct CodecCompressParams
{
...
    /* The following fields only exist for QuickTime 2.1 and greater */
    UInt16                      majorSourceChangeSeed;
    UInt16                      minorSourceChangeSeed;
    CDSequenceDataSourcePtr     sourceData;

    /* The following fields only exit for QuickTime 2.5 and greater */
    long                        preferredPacketSizeInBytes;
};
```

Field description

`majorSourceChangeSeed`

Contains an integer value that is incremented each time a data source is added or removed. This provides a fast way for a codec to know when it needs to redetermine which data source inputs are available.

`minorSourceChangeSeed`
: Contains an integer value that is incremented each time a data source is added or removed, or the data contained in any of the data sources changes. This provides a way for a codec to know if the data available to it has changed.

`sourceData`
: Contains a pointer to a `CDSequenceDataSource` structure (page 4-11). This structure contains a linked list of all data sources. Because each data source contains a link to the next data source, a codec can access all data sources from this field.

`preferredPacketSizeInBytes`
: Specifies the preferred packet size for data.

The Decompression Parameters Structure

Several fields have been added to the decompression parameters structure (`CodecDecompressParams`) originally documented in Inside Macintosh: QuickTime Components. In addition, several new flags exist that could be set in the `codecConditions` field. The first two fields listed below (frameTime, reserved) replace the last field (reserved) documented in *IM: QT Components*.

```
struct CodecDecompressParams
{
...
/* The following fields only exist for QuickTime 2.0 and greater */
    ICMFrameTimePtr     frameTime;      /* banddecompress */
    long                reserved[1];

    SInt8               matrixFlags;
    SInt8               matrixType;
    Rect                dstRect;    /*  only valid for
simple transforms */

/* The following fields only exist for QuickTime 2.1 and greater */
    UInt16              majorSourceChangeSeed;
    UInt16              minorSourceChangeSeed;
    CDSequenceDataSourcePtr     sourceData;

    RgnHandle       maskRegion;
```

```
/* The following fields only exist for QuickTime 2.5 and greater */
    OSType              **wantedDestinationPixelTypes;/* Handle to
                                        0-terminated list of OSTypes */

    long                screenFloodMethod;
    long                screenFloodValue;
    short               preferredOffscreenPixelSize;
};
```

Field descriptions

`frameTime` Contains a pointer to an `ICMFrameTime` structure (page 4-10). This structure contains a frame's time information for scheduled asynchronous decompression operations.

`matrixFlags` Flags specifying information about the transformation matrix. Currently, can be 0 or one of the following:

```
enum {
    matrixFlagScale2x = 1L<<7,
    matrixFlagScale1x = 1L<<6,
    matrixFlagScaleHalf = 1L<<5
};
```

`matrixType` Contains the type of the transformation matrix, as returned by `GetMatrixType()`. (For additional information refer to Inside Macintosh: QuickTime, p. 2-342).

`dstRect` The destination rectangle. The result of the source rectangle (`srcRect`) transformed by the transformation matrix (`matrix`).

`majorSourceChangeSeed` Contains an integer value that is incremented each time a data source is added or removed. This provides a fast way for a codec to know when it needs to redetermine which data source inputs are available.

`minorSourceChangeSeed` Contains an integer value that is incremented each time a data source is added or removed, or the data contained in any of the data sources changes. This provides a way for a codec to know if the data available to it has changed.

`sourceData` Contains a pointer to a `CDSequenceDataSource` structure (page 4-11). This structure contains a linked list of all data

sources. Because each data source contains a link to the next data source, a codec can access all data sources from this field.

`maskRegion` If the `maskRegion` field is not `nil`, it contains a QuickDraw region which is equivalent to the bit map contained in the `maskBits` field. For some codecs, using the QuickDraw region may be more convenient than the mask bit map.

`wantedDestinationPixelTypes`
Filled in by the codec during `ImageCodecPreDecompress`. Contains a handle to a zero-terminated list of non-RGB pixels that the codec can decompress to. Leave set to `nil` if the codec does not support non-RGB pixel spaces. The ICM copies this data structure, so it is up to the codec to dispose of it later. Since the predecompress call can be called often, it is suggested that codecs allocate this handle during the Open routine and dispose of it during the Close routine.

`screenFloodMethod` For codecs that require key-color flooding. One of:

```
enum {
    kScreenFloodMethodNone = 0,
    kScreenFloodMethodKeyColor = 1,
    kScreenFloodMethodAlpha = 2
};
```

`screenFloodValue` If screenFloodMethod is `kScreenFloodMethodKeyColor`, contains the index of the color that should be used to flood the image area on screen when an refresh occurs. This is valid for both indexed and direct screen devices (e.g., for 16 bpp devices, it should contain the 5-5-5 RGB value). If `screenFloodMethod` is `kScreenFloodMethodAlpha`, contains the value that the alpha-channel should be flooded with.

`preferredOffscreenPixelSize`
Should be filled in `ImageCodecPreDecompress` with the preferred depth of an offscreen buffer should the ICM have to create one. It is not guaranteed that an offscreen will actually be of this depth. A codec should still be sure to specify what depths it can decompress to by using the capabilities field. A codec might use this field if if was capable of decompressing to several depths, but was faster decompressing to a particular depth.

codecConditions flags

Several new flags exist that can be set in the `codecConditions` parameter. They are:

```
enum {
    codecConditionFirstScreen       = 1L << 12,
    codecConditionDoCursor          = 1L << 13,
    codecConditionCatchUpDiff       = 1L << 14,
    codecConditionMaskMayBeChanged  = 1L << 15,
    codecConditionToBuffer          = 1L << 16
};
```

`codecConditionFirstScreen`

Indicates when the codec is decompressing an image to the first of multiple screens. That is, if the decompressed image crosses multiple screens, then the codec can look at this flag to determine if this is the first time an image is being decompressed for each of the screens to which it is being decompressed.

A codec that depends on the `maskBits` field of `decompressParams` being a valid `regionHandle` on `CDPreDecompress` needs to know that in this case it is not able to clip images since the region handle is only passed for the first of the screens; clipping would be incorrect for the subsequent screen for that image.

`codecConditionDoCursor`

Set to 1 if the decompressor component should shield and unshield the cursor for the current decompression operation. This flag will only be set if the codec has indicated its ability to handle cursor shielding by setting the `codecCanShieldCursor` flag in the capabilities field during `CDPreDecompress`.

`codecConditionCatchUpDiff`

Indicates if the current frame is a “catch up” frame. Set this flag to 1 if the current frame is a catch-up frame. Note that you must also set the `codecFlagCatchUpDiff` flag to 1. This may be useful to decompressors which can drop frames when playback is falling behind.

`codecConditionMaskMayBeChanged`

The Image Compression Manager has always included

support for decompressors that could provide a bit mask of pixels that were actually drawn when a particular frame was decompressed. If a decompressor can provide a bit mask of pixels that changed, the Image Compression Manager transfers to the screen only the pixels that actually changed.

QuickTime 2.1 extends this capability by adding a new condition flag (`codecConditionMaskMayBeChanged`) to the `conditionFlags` field of the decompression parameters structure. The decompressor should only write back the mask only when this flag is set. The flag is used only by the `ImageCodecBandDecompress` function (page 4-23).

`codecConditionToBuffer`

Set to 1 if the current decompression operation is decompressing into an offscreen buffer.

Functions

ImageCodecExtractAndCombineFields

Performs field operations on video data. It allows fields from two separate images, compressed in the same format, to be combined in to a new compressed frame. Typically the operation results in an image of identical quality to the source images. Fields of a single image may also be duplicated or reversed by this function.

```
pascal ComponentResult ImageCodecExtractAndCombineFields (
                    ComponentInstance ci,
                    long fieldFlags,
                    void *data1,
                    long dataSize1,
                    ImageDescriptionHandle desc1,
                    void *data2,
                    long dataSize2,
                    ImageDescriptionHandle desc2,
```

```
        void *outputData,
        long *outDataSize,
        ImageDescriptionHandle descOut);
```

`ci` Specifies the image compressor component for the request.

`fieldFlags` Flags specifying the operation to be performed. A correctly formed request will specify two input fields, mapping one to the odd output field and the other to the even output field. The following flags are defined:

`evenField1ToEvenFieldOut`
: Maps the even field specified by the `data1` parameter to the even output field.

`evenField1ToOddFieldOut`
: Maps the even field specified by the `data1` parameter to the odd output field.

`oddField1ToEvenFieldOut`
: Maps the odd field specified by the `data1` parameter to the even output field.

`oddField1ToOddFieldOut`
: Maps the odd field specified by the `data1` parameter to the odd output field.

`evenField2ToEvenFieldOut`
: Maps the even field specified by the `data2` parameter to the even output field.

`evenField2ToOddFieldOut`
: Maps the even field specified by the `data2` parameter to the odd output field.

`oddField2ToEvenFieldOut`
: Maps the odd field specified by the `data2` parameter to the even output field.

`oddField2ToOddFieldOut`
: Maps the odd field specified by the `data2` parameter to the odd output field.

`data1` A pointer to a buffer containing the compressed image data for the first input field.

`dataSize1` Specifies the size of the `data1` buffer.

desc1	An image description structure describing the format and characteristics of the data in the data1 buffer.
data2	A pointer to a buffer containing the compressed image data for the second input field. Set to nil if the requested operation uses only one input frame.
dataSize2	Specifies the size of the data2 buffer. Set to 0 if the requested operation uses only one input frame.
desc2	An image description structure describing the format and characteristics of the data in the data2 buffer. Set to nil if the requested operation uses only one input frame.
outputData	A pointer to a buffer to receive the resulting frame.
outDataSize	On output this parameter returns the actual size of the new compressed image data.
descOut	Specifies the desired format of the resulting frames. Typically this is the same format specified by the desc1 and desc2 parameters.

DISCUSSION

This codec routine implements the the functionality of the ImageFieldSequenceExtractCombine function described in Chapter 3, "Image Compression Manager." If your codec is capable of separately compressing both fields of a video frame, you should incorporate support for this function.

Your codec must ensure that it understands the image formats specified by desc1 and desc2 parameters, as these may not be the same as the codec's native image format. The image format specified by the descOut parameter will be the same as the codec's native image format.

The component selector for this function is:

```
kImageCodecExtractAndCombineFieldsSelect    = 0x0015
```

ImageCodecPreDecompress

Your component receives an `ImageCodecPreDecompress` call before decompressing an image or sequence of frames.

```
pascal ComponentResult ImageCodecPreDecompress(
                    ComponentInstance ci,
                    CodecDecompressParams *params);
```

`ci` Specifies the image decompressor component for the request.

`params` Contains a pointer to a decompression parameters structure. See "The Decompression Parameters Structure" (*Inside Macintosh: QuickTime Components*, page 4-46), and "The Decompression Parameters Structure" (page 4-15) of this volume for a complete description.

DISCUSSION

If your decompressor component supports scheduled asynchronous decompression operations, be sure to set the `codecCanAsyncWhen` flag to 1 in the `flags` field of your component's compressor capabilities structure. If you set `codecCanAsyncWhen` you must also set `codecCanAsync`. Codecs that support scheduled asynchronous decompression are strongly advised to also set the `codecCanShieldCursor` flag.

If your decompressor component uses a secondary hardware buffer for its images, be sure to set the `codecHasVolatileBuffer` flag to 1 in the `flags` field of your component's compressor capabilities structure.

If your decompressor component is used solely as a transfer codec and uses the `CDCodecNewImageBufferMemory` call to create an offscreen buffer that is really onscreen, your codec will need to set the `codecImageBufferIsOnScreen` flag to 1.

See the section "The Compressor Capability Structure" (page 4-12) for more information about these flags.

ImageCodecBandDecompress

Your component receives an `ImageCodecBandDecompress` call to decompress a frame.

```
pascal ComponentResult ImageCodecBandDecompress(
                    ComponentInstance ci,
                    CodecDecompressParams *params);
```

`ci` Specifies the image decompressor component for the request.

`params` Contains a pointer to a decompression parameters structure. See "The Decompression Parameters Structure" (*Inside Macintosh: QuickTime Components*, page 4-46), and "The Decompression Parameters Structure" (page 4-15) of this volume for a complete description.

DISCUSSION

For scheduled asynchronous decompression operations, the Image Compression Manager supplies a reference to an `ICMFrameTime` structure in this function's decompression parameters structure parameter. The `ICMFrameTime` structure (page 4-10) contains time information governing the scheduled decompression operation, including the time at which the frame must be displayed. For synchronous or immediate asynchronous decompress operations, the frame time is set to `nil`.

When your component has finished the decompression operation, it must call the completion function. In the past, for asynchronous operations, your component called that function directly. For scheduled asynchronous decompression operations, your component should call the Image Compression Manager's `ICMDecompressComplete` function(page 4-36).

If your component set the `codecCanAsyncWhen` flag in pre-decompress but cannot support scheduled asynchronous decompression for a given frame, it must return an error code of `codecCantWhenErr`. If your component's queue is full, it should return an error code of `codecCantQueueErr`.

ImageCodecFlush

Empties a image decompressor component's input queue of pending shceduled frames.

```
pascal ComponentResult ImageCodecFlush(
                    ComponentInstance ci);
```

`ci` Specifies the image decompressor component for the request.

DISCUSSION

Your component receives the `ImageCodecFlush` function whenever the Image Compression Manager needs to cancel the display of all scheduled frames.

Your decompressor should empty its queue of scheduled asynchronous decompression requests. For each request, your component must call the `ICMDecompressComplete` function. Be sure to set the `err` parameter to –1, indicating that the request was canceled. Also, you must set both the `codecCompletionSource` and `codecCompletionDest` flags to 1. Only decompressor components that support scheduled asynchronous decompression will recieve this call.

SPECIAL CONSIDERATIONS

Your component's `ImageCodecFlush` function may be called at interrupt time.

ImageCodecSetTimeCode

Sets the timecodefor the next frame that is to be decompressed.

```
pascal OSErr ImageCodecSetTimeCode (
                    ComponentInstance ci,
                    const TimeCodeDef *timeCodeFormat,
                    const TimeCodeTime *timeCodeTime);
```

`ci` Specifies the image decompressor component for the request.

`timeCodeFormat`
: Contains a pointer to a timecode definition structure. This structure contains the timecode definition information for the next frame to be decompressed.

`timeCodeTime`
: Contains a pointer to a timecode record structure. This structure contains the time value for the next frame in the current sequence.

DISCUSSION

Your component receives `CDCodecSetTimeCode` function whenever an application calls the Image Compression Manager's `SetDSequenceTimeCode` function. That function allows an application to set the timecode for a frame that is to be decompressed.

The timecode information you receive applies to the next frame to be decompressed and is provided to the decompressor in the `CDBandDecompress` function.

ImageCodecIsImageDescriptionEquivalent

Compares image descriptions.

```
pascal ComponentResult ImageCodecIsImageDescriptionEquivalent (
                    ComponentInstance ci,
                    ImageDescriptionHandle newDesc,
                    Boolean *equivalent);
```

`ci`
: Specifies the image compressor component for the request.

`newDesc`
: Contains a handle to the image description structure that describes the compressed image.

`equivalent`
: Contains a pointer to a Boolean value. If the `ImageDescriptionHandle` provided in the `newDesc` parameter is equivalent to the image description currently in use by the image sequence, this value is set to `true`. If the

`ImageDescriptionHandle` is not equivilent, and therefore a new image sequence must be created to display an image using the new image description, this value is set to `false`.

DISCUSSION

Your component receives the `ImageCodecIsImageDescriptionEquivalent` request whenever an application calls the Image Compression Manager's `CDSequenceEquivalentImageDescription` function (page 3-21). Implementing this function can significantly improve playback of edited video sequences using your codec. For example, if two sequences are compressed at different quality levels and are edited together they will have different image descriptions because their quality values will be different. This will force QuickTime to use two separate decompressor instances to display the images. By implementing this function your decompressor can tell QuickTime that differences in quality levels don't require separate decompressors. This saves memory and time thus improves performance.

SPECIAL CONSIDERATIONS

The current image description is not passed in this function because the Image Compression Manager assumes the codec has already made copies of all relevant data fields from the current image description during the `ImageCodecPreDecompress` call.

ImageCodecNewMemory

Requests codec-allocated memory. Some hardware codecs may have on-board memory which can be used to store compressed and/or decompressed data. `ImageCodecNewMemory` makes this memory available for use by clients of the codec. Some software codecs may be able to optimize thier performance by having more control over memory allocation. `ImageCodecNewMemory` makes this control available.

```
pascal ComponentResult ImageCodecNewMemory (
                    ComponentInstance ci,
                    Ptr *data,
```

```
                    Size dataSize,
                    long dataUse,
                    ICMMemoryDisposedUPP memoryGoneProc,
                    void *refCon);
```

`ci` Specifies the image decompressor component for the request.

`data` Returns a pointer to the allocated memory.

`dataSize` Specifies the desired size of the `data` buffer.

`dataUse` A code that indicates how the memory is to be used. For example, the memory may be used to store compressed data before it's displayed, mask plane data, or decompressed data.

If there is no benefit to storing a particular kind of data in codec memory, the codec should refuse the request for the memory allocation. The defined values are for data use are:

0x00000001	Memory will be used for holding compressed image data.
0x00000002	Memory will be used for an offscreen image buffer.

`memoryGoneProc`

A pointer to a function that will be called before disposing of the memory allocated by a codec. Your callback function must be in the following form:

```
pascal void (*ICMMemoryDisposedProcPtr)
                (Ptr memoryBlock, void *refcon);
```

This function must be called if the memory block is to be disposed of by the codec instead of by `ImageCodeDisposeMemory`. For example, this would occur if the codec is closed and still has memory allocation outstanding or if the memory is required to complete another operation. The `memoryGoneProc` must not be called at interrupt time.

`refCon` Contains a reference constant value that your codec must pass to the `memoryGoneProc` function.

DISCUSSION

Your component receives the `ImageCodecNewMemory` request whenever an application calls the Image Compression Manager's `CDSequenceNewMemory` function (page 3-22).

SPECIAL CONSIDERATIONS

The Image Compression Manager does not currently track memory allocations. When a compressor or decompressor component instance is closed, it must ensure that all blocks allocated by that instance are disposed (and call the `ICMMemoryDisposeUPP`). If your codec does not currently have free memory for compression frame data, but will soon , you can return `codecMemoryFullPleaseWait` to indicate this fact.

ImageCodecNewImageBufferMemory

Requests the codec to allocate memory for an offscreen buffer of non-RGB pixels. This call is used to support a codec decompressing into a non-RGB buffer. The transfer codec is responsible for defining the offscreen and transferring the image from the offscreen to the destination.

```
pascal ComponentResult ImageCodecNewImageBufferMemory(
                    ComponentInstance ci,
                    CodecDecompressParams *params,
                    long flags,
                    ICMMemoryDisposedUPP memoryGoneProc,
                    void *refCon)
```

`ci` Specifies the image decompressor component for the request.

`params` Contains a pointer to a decompression parameters structure. See "The Decompression Parameters Structure" (*Inside Macintosh: QuickTime Components*, page 4-46), and "The Decompression Parameters Structure" (page 4-15) of this volume for a complete description. Your codec must fill in the dstPixMap.baseAddr and the dstPixMap.rowBytes fields in this structure.

`flags` Currently, this parameter is always set to 0.

`memoryGoneProc`

A pointer to a function that will be called before disposing of the memory allocated by a codec. Your callback function must be in the following form:

```
pascal void (*ICMMemoryDisposedProcPtr)
             (Ptr memoryBlock, void *refcon);
```

This function must be called if the memory block is to be disposed of by the codec instead of by `ImageCodeDisposeMemory`. For example, this would occur if the codec is closed and still has memory allocation outstanding or if the memory is required to complete another operation. The `memoryGoneProc` must not be called at interrupt time.

`refCon` Contains a reference constant value that your codec must pass to the `memoryGoneProc` function.

DISCUSSION

Your component receives the `ImageCodecNewImageBufferMemory` request whenever another codec has requested a non-RGB offscreen buffer of the with a type of your component's subtype. See "Accelerated Video Support" (page 4-5) for more information.

SPECIAL CONSIDERATIONS

The Image Compression Manager does not currently track memory allocations. When a compressor or decompressor component instance is closed, it must ensure that all blocks allocated by that instance are disposed (and call the `ICMMemoryDisposeUPP`).

ImageCodecDisposeMemory

Disposes codec-allocated memory.

```
pascal ComponentResult ImageCodecDisposeMemory (
                    ComponentInstance ci,
                    Ptr data);
```

ci	Specifies the image compressor component for the request
data	Points to the previously allocated memory block.

DISCUSSION

Your component receives the `ImageCodecDisposeMemory` request whenever an application calls the Image Compression Manager's `CDSequenceDisposeMemory` function (page 3-24).

SPECIAL CONSIDERATIONS

When a codec instance is closed, it must ensure that all blocks allocated by that instance are disposed (and call the `ICMMemoryDisposeUPP`).

ImageCodecRequestSettings

Displays a dialog containing codec-specific compression settings.

```
pascal ComponentResult ImageCodecRequestSettings (
                    ComponentInstance ci,
                    Handle settings,
                    Rect *rp,
                    ModalFilterUPP filterProc);
```

ci	Specifies the image compressor component for the request.
settings	A handle of data specific to the codec. If the handle is empty, the codec should use its default settings.
rp	A pointer to a rectangle giving the coordinates of the Standard Compression dialog in global screen coordinates. The codec can use this to position its dialog box in the same area of the screen.
filterProc	A pointer to a modal dialog filter procedure that the codec must either pass to the `ModalDialog` function or call at the beginning of the codec dialog filter. This procedure gives the calling application and Standard Compression dialog a chance to process update events.

DISCUSSION

The `ImageCodecRequestSettings` function allows the display of a dialog box of additional compression settings specific to the codec. These settings are stored in a `settings` handle. The codec can store any data in any format it wants in the `settings` handle and resize it accordingly. It should store some type of tag or version information that it can use to verify that the data belongs to the codec. The codec should not dispose of the handle.

ImageCodecGetSettings

Returns the settings chosen by the user.

```
pascal ComponentResult ImageCodecGetSettings (
                    ComponentInstance ci,
                    Handle settings);
```

`ci` Specifies the image compressor component for the request.

`settings` A handle that the codec should resize and fill in with the current internal settings. If there are no current internal settings, resize it to 0. Don't dispose of this handle.

DISCUSSION

The `ImageCodecGetSettings` function allows a codec to return it's current private settings. From this function, the codec should return its current internal settings. If there are no current settings or the settings are the same as the defaults, the codec can set the handle to `nil`.

ImageCodecSetSettings

Sets the settings of the optional dialog box.

```
pascal ComponentResult ImageCodecSetSettings (
                    ComponentInstance ci,
                    Handle settings);
```

ci	Specifies the image compressor component for the request. Applications obtain this reference from the Component Manager's OpenComponent function.
settings	A handle to internal settings originally returned by either ImageCodecRequestSettings or ImageCodecGetSettings. The codec should set its internal settings to match those of the settings handle. Because the codec does not own the handle, it should not dispose of it and should copy only its contents, not the handle itself. If the settings handle passed is empty, the codec should sets its internal settings to a default state.

DISCUSSION

The ImageCodecSetSettings function allows a codec to return it's private settings. Set the codec's internal settings to the state specified in the settings handle. The codec should always check the validity of the contents of the handle so that invalid settings are not used.

ImageCodecHitTestData

This routine is called when the application calls PtInDSequenceData. It returns a Boolean indicating whether or not the specified point is contained within the specified image data.

```
pascal ComponentResult ImageCodecHitTestData(
                    ComponentInstance ci,
                    ImageDescriptionHandle desc,
                    void *data,
                    Size dataSize,
                    Point where,
                    Boolean *hit)
```

ci	Specifies the image decompressor component for the request.
desc	Contains an ImageDescriptionHandle for the image data pointed to by the data param.
data	Pointer to compressed data in the format specified by the desc param.

dataSize	Size of the compressed data referred to by the data param.
where	A QuickDraw Point (0,0) based at the top-left corner of the image.
hit	A pointer to a Boolean. The Boolean should be set to true if the point specified by the where parameter is contained within the compressed image data specified by the data param.

DISCUSSION

The ImageCodecHitTestData function allows the calling application to perform hit testing on compressed data. The codec should set the hit parameter to true if the compressed data contains data at the point specified by the where parameter. The hit parameter should be set to false if the specified point falls within a blank portion of the image.

ImageCodecGetMaxCompressionSizeWithSources

Your codec receives the request when an application calls the Image Compression Manager's GetCSequenceMaxCompressionSize function. The caller uses this function to determine the maximum size the data will be compressed to for a given image and set of data sources.

```
pascal ComponentResult ImageCodecGetMaxCompressionSizeWithSources(
                    ComponentInstance ci,
                    PixMapHandle src,
                    const Rect *srcRect,
                    short depth,
                    CodecQ quality,
                    CDSequenceDataSourcePtr sourceData,
                    long *size)
```

ci	Specifies the image decompressor component for the request.
src	Contains a handle to the source image. The source image is stored in a pixel map structure. Applications use the size information you return to allocate buffers for more than one image. Consequently, your compressor should not consider the

contents of the image when determining the maximum compressed size. Rather, you should consider only the quality level, pixel depth, and image size.

This parameter may be set to nil. In this case the application hs not supplied a source image – your component should use the other parameters to determine the characteristics of the image to be compressed.

`srcRect` Contains a pointer to a rectangle defining the portion of the source image to compress.

`depth` Specifies the depth at which the image is to be compressed. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 33, 34, 36, and 40 indicate 1-bit, 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images.

`quality` Specifies the desired compression image quality. See the chapter "Image Compression Manager" in *Inside Macintosh: QuickTime* for valid values.

`sourceData` Contains a pointer to a `CDSequenceDataSource` structure (page 4-11). This structure contains a linked list of all data sources. Because each data source contains a link to the next data source, a codec can access all data sources from this field

`size` Contains a pointer to a field to receive the maximum size, in bytes, of the compressed image.

DISCUSSION

The `ImageCodecGetMaxCompressionSizeWithSources` function is similar in purpose to the `ImageCodecGetMaxCompressionSize` function documented in *Inside Macintosh: QuickTimeComponents* (page 4-55). This function, however, also considers data sources that the codec may compress with the image.

ImageCodecSourceChanged

Your codec receives this notification that one of the data sources has changed when an application calls the Image Compression Manager's `CDSequenceSetSourceData` or `CDSequenceChangedSourceData` functions.

```
pascal ComponentResult ImageCodecSourceChanged(
                    ComponentInstance ci,
                    UInt32 majorSourceChangeSeed,
                    UInt32 minorSourceChangeSeed,
                    CDSequenceDataSourcePtr sourceData)
```

`ci` Specifies the image decompressor component for the request.

`majorSourceChangeSeed`
Contains an integer value that is incremented each time a data source is added or removed. This provides an easy way for a codec to know when it needs to redetermine which data source inputs are available.

`minorSourceChangeSeed`
Contains an integer value that is incremented each time a data source is added or removed, or the data contained in any of the data sources changes. This provides a way for a codec to know if the data available to it has changed.

`sourceData` Contains a pointer to a `CDSequenceDataSource` structure (page 4-11). This structure contains a linked list of all data sources. Because each data source contains a link to the next data source, a codec can access all data sources from this field.

DISCUSSION

This routine is provided to notify the codec component that one of the data sources has changed.

Image Compression Manager Utility Functions

ICMShieldSequenceCursor

Hides the cursor during decompression operations.

```
pascal OSErr ICMShieldSequenceCursor (ImageSequence seqID);
```

`seqID` Identifies the sequence for which to shield the cursor.

DISCUSSION

Your component may call the `ICMShieldSequenceCursor` function to manage the display of the cursor during decompression operations.

For correct image display behavior, the cursor must be shielded (hidden) during decompression. By default, the Image Compression Manager handles the cursor for you, hiding it at the beginning of a decompression operation and revealing it at the end.

With the advent of scheduled asynchronous decompression, however, the Image Compression Manager cannot do as precise a job of managing the cursor, because it does not know exactly when scheduled operations actually begin and end. While the Image Compression Manager can still manage the cursor, it must hide the cursor when each request is queued, rather than when the request is serviced. This may result in the cursor remaining hidden for long periods of time.

In order to achieve better cursor behavior, you can choose to manage the cursor in your decompressor component. If you so choose, you can use the `ICMShieldSequenceCursor` function to hide the cursor—the Image Compression Manager displays the cursor when you call the `ICMDecompressComplete` function. In this manner, the cursor is hidden only when your component is decompressing and displaying the frame.

SPECIAL CONSIDERATIONS

This function may be called at interrupt time.

ICMDecompressComplete

Signals the completion of a decompression operation.

```
pascal void ICMDecompressComplete (
                    ImageSequence seqID,
                    OSErr err,
                    short flag,
                    ICMCompletionProcRecordPtr completionRtn);
```

`seqID` Identifies the frame's sequence.

`err` Indicates whether the operation succeeded or failed. Set this parameter to 0 for successful operations. For failed operations, set the error code appropriate for the failure. For canceled operations (for example, when the Image Compression Manager calls your component's `ImageCodecFlush` function), set this parameter to –1.

`flag` Completion flags. Note that you may set more than one of these flags to 1. The following flags are defined:

`codecCompletionSource`
: Your component is done with the source buffer. Set this flag to 1 when you are done with the processing associated with the source buffer.

`codecCompletionDest`
: Your component is done with the destination buffer. Set this flag to 1 when you are done with the processing associated with the destination buffer.

`codecCompletionDontUnshield`
: Set this flag to 1 when you do not want the Image Compression Manager to unshield the cursor as it normally would. Only codecs that are completely managing the cursor themselves should set this flag. See the section "Hardware Cursors" (page 4-4) for more information.

`completionRtn`

Contains a pointer to a completion function structure. That structure identifies the application's completion function, and contains a reference constant associated with the frame.

Your component obtains the completion function structure as part of the decompression parameters structure provided by the Image Compression Manager at the start of the decompression operation.

DISCUSSION

Your component must call this function at the end of decompression operations.

SPECIAL CONSIDERATIONS

Prior to QuickTime 2.0, decompressor components called the application's completion function directly. For compatibility, that method is still supported except for scheduled asynchronous decompression operations, which must use the `ICMDecompressComplete` call. Newer decompressors should always use `ICMDecompressComplete` rather than calling the completion function directly regardless of the type of decompression operation.

Image Transcoder Components

Contents

This chapter is for those writing image transcoders. To use the transcoders refer to the chapter "Image Compression Manager". Before QuickTime 2.5, if you needed to convert compressed image data into another compressed image format, it was necessary to decompress the compressed image data to RGB pixels, and then compress the RGB pixels into the new format. For some types of compressed image data, it is possible to convert directly from one compressed format to another. This direct conversion process is called **image transcoding**.

This chapter is divided into the following sections:

- "About Image Transcoding" provides a general introduction to image transcoding and components of this type
- "Using Image Transcoder Components" describes how QuickTime uses these components
- "Creating an Image Transcoder Component" describes how to create one of these components
- "Reference to Image Transcoder Components" presents detailed information about the functions that are supported by these components

About Image Transcoding

Transcoding has two advantages over the decompress then recompress approach to converting the format of compressed data. First, the operation is usually substantially faster since much of the data can be copied directly from the source image data format to the destination image data format. Secondly, the operation is usually more accurate because decompressing and recompressing provides two steps for introducing rounding and quantization errors. By directly transcoding, opportunities for small errors are substantially reduced.

Image Transcoding Support

QuickTime's image transcoding support is contained within the Image Compression Manager. Image transcoding can be invoked either explicitly, using new API's in the Image Compression Manager, or implicitly, by using existing routines for decompressing images.

QuickTime's support for decompressing images has been enhanced so that if a request is issued to decompress an image, but no image decompressor component is installed for that image format, QuickTime will attempt to locate an image transcoder to convert the image data into a supported format. This transcoding is performed transparently to the calling application. This automatic image transcoding is supported for both QuickTime movies and compressed image data stored in QuickDraw pictures.

QuickTime also provides an API for applications to transcode images. These API's make it possible for any application to take compressed image data and transcode it into another format. This capability is useful for applications that create QuickTime movies by combining segments of other QuickTime movies. These applications often convert the format of the compressed image data by decompressing the image and then recompressing it to the new format. If no other processing is to be performed on the compressed data, you can use an image transcoder to increase the speed and fidelity of the operation.

As with most other services in QuickTime, the details of image transcoding are handled by components. The Image Compression Manager uses image transcoder components to perform both implicit and explicit image transcoding. Application developers that perform image transcoding interact with the Image Compression Manager, not directly with the image transcoder components themselves. The Image Compression Manager takes care of the details of working with image transcoder components. If you want to add new image transcoding operations to QuickTime, you can write an image transcoder component.

Using Image Transcoder Components

The Image Compression Manager uses an image sequence when compressing or decompressing data. An image sequence allows QuickTime to make certain optimizations because it knows that a similar operation will be repeated multiple times (that is, images will be repeatedly compressed to the same image data format). Similarly, the Image Compression Manager's support for image transcoding is based on an image transcoding sequence. The image transcode sequence identifier is an opaque value as shown below.

```
typedef long ImageTranscodeSequence;
```

All transcoding functions are described in Chapter 3, “Image Compression Manager.” Image transcoder component functions are described later in this chapter.

To create an image transcoding sequence, use the `ImageTranscodeSequenceBegin` function. To transcode a frame of image data, use the `ImageTranscodeFrame` function. The caller of this routine is responsible for disposing of the transcoded data returned by `ImageTranscodeFrame` using the `ImageTranscodeDisposeFrameData` routine.

When the transcoded image data returned by `ImageTranscodeFrame` is no longer needed, call `ImageTranscodeDisposeFrameData` to dispose of the data. When an image transcoding sequence is complete, use `ImageTranscodeSequenceEnd` to dispose of the image transcoding sequence.

Creating an Image Transcoder Component

It is only necessary to understand image transcoder components if you are writing an image transcoder. To perform image transcoding, you should use the services provided by the Image Compression Manager.

Image transcoder components are standard Component Manager components. See *Inside Macintosh: More Macintosh Toolbox* for details on creating components.

Image transcoder components have a type of ‘`imtc`’ as defined below.

```
enum {
    ImageTranscoderComponentType = 'imtc'
};
```

The sub-type field of the component defines the compressed image data format that the transcoder accepts as an input. The manufacturer field of the component defines the compressed image data format that the transcoder generates as output. For example, a trancoder from Motion JPEG Format A to Motion JPEG Format B would have a subtype of `'mjpg'` and a manufacturer code of `'mjpb'`. No component-specific flags are currently defined for transcoders; they should be set to 0. Each transcoder component function is described later in this chapter.

Example Image Transcoder Component

The following example code shows an image transcoder component. It converts an imaginary compressed data format 'bgr' to uncompressed RGB pixels. The transcoding process simply copies the source data to the destination and inverts each byte in the process. This example shows the format of how an image transcoder might work without getting into the details of a particular image transcoding operation.

```
#include <ImageCompression.h>
pascal ComponentResult main(ComponentParameters *params, Handle storage
);
pascal ComponentResult TestTranscoderBeginSequence (Handle storage,
ImageDescriptionHandle srcDesc, ImageDescriptionHandle *dstDesc, void
*data, long dataSize);

pascal ComponentResult TestTranscoderConvert (Handle storage, void
*srcData, long srcDataSize, void **dstData, long *dstDataSize);

pascal ComponentResult TestTranscoderDisposeData (Handle storage, void
*dstData);

pascal ComponentResult TestTranscoderEndSequence (Handle storage);

pascal ComponentResult main(ComponentParameters *params, Handle storage )
{
    ComponentFunctionUPP proc = nil;
    ComponentResult err = noErr;

    switch (params->what) {
        case kComponentOpenSelect:
        case kComponentCloseSelect:
            break;
        case kImageTranscoderBeginSequenceSelect:
            proc = (ComponentFunctionUPP) TestTranscoderBeginSequence;
            break;
        case kImageTranscoderConvertSelect:
            proc = (ComponentFunctionUPP)TestTranscoderConvert;
            break;
        case kImageTranscoderDisposeDataSelect:
            proc = (ComponentFunctionUPP) TestTranscoderDisposeData;
            break;
```

```
        case kImageTranscoderEndSequenceSelect:
            proc = (ComponentFunctionUPP) TestTranscoderEndSequence;
            break;
        default:
            err = badComponentSelect;
            break;
    }

    if (proc)
        err = CallComponentFunctionWithStorage(storage,
                        params, proc);

    return err;
}

pascal ComponentResult TestTranscoderBeginSequence (Handle storage,
ImageDescriptionHandle srcDesc, ImageDescriptionHandle *dstDesc, void
*data, long dataSize)
{
    *dstDesc = srcDesc;
    HandToHand((Handle *)dstDesc);
    (***dstDesc).cType = 'raw ';

    return noErr;
}

pascal ComponentResult TestTranscoderConvert (Handle storage, void
*srcData, long srcDataSize, void **dstData, long *dstDataSize)
{
    Ptr p;
    OSErr err;

    if (!srcDataSize)
        return paramErr;

    p = NewPtr(srcDataSize);
    err = MemError();
    if (err) return err;
    {
    Ptr p1 = srcData, p2 = p;
    long counter = srcDataSize;
```

```
    while (counter--)
        *p2++ = ~*p1++;
    }

    *dstData = p;
    *dstDataSize = srcDataSize;

    return noErr;
}

pascal ComponentResult TestTranscoderDisposeData (Handle storage, void
*dstData)
{
    DisposePtr((Ptr)dstData);

    return noErr;
}

pascal ComponentResult TestTranscoderEndSequence (Handle storage)
{
    return noErr;
}
```

Image Transcoder Components Reference

Functions

QuickTime 2.5 provides four image transcoder component functions.

ImageTranscoderBeginSequence

Initiates an image transcoding sequence and specifies the input data format.

```
pascal ComponentResult ImageTranscoderBeginSequence (
                    ImageTranscoderComponent itc,
                    ImageDescriptionHandle srcDesc,
                    ImageDescriptionHandle *dstDesc,
                    void *data,
                    long dataSize);
```

itc	The image transcoder component.
srcDesc	The image description for the source compressed image data.
dstDesc	Returns a new image description.
data	First frame of data to be transcoded (may be nil).
dataSize	Size of compressed image data pointed to by the data.

DISCUSSION

The `ImageTranscoderBeginSequence` function specifies the format of source compressed image data in the `srcDesc` parameter. The image transcoder should allocate a new image description and return it in the `dstDesc` parameter. The new image description should be a completely filled out image description which is sufficient for correctly decompressing the data generated by subsequent calls to `ImageTranscoderConvert`.

ImageTranscoderConvert

Performs image transcoding operations.

```
pascal ComponentResult ImageTranscoderConvert (
                    ImageTranscoderComponent itc,
                    void *srcData,
                    long srcDataSize,
                    void **dstData,
                    long *dstDataSize);
```

`itc`	The image transcoder component.
`srcData`	Contains a pointer to the source compressed image data to transcode.
`srcDataSize`	Indicates the size of the source image data, in bytes.
`dstData`	Returns a pointer to the transcoded data.
`dstDataSize`	Returns the size of the transcoded data, in bytes.

DISCUSSION

The image transcoder component is responsible for allocating storage for the transcoded data, transcoding the data, and returning a pointer to the transcoded data in the `dstData` parameter. The size of the transcoded data in bytes should be returned in the `dstDataSize` parameter. The caller is responsible for disposing of the transcoded data using the `ImageTranscoderDisposeData` function.

The memory allocated to store the transcoded image data must not be in an unlocked handle. Even if the image transcoding operation can be performed in place, the transcoded data must be placed in a separate block of memory from the source data. The image transcoder component must not write back into the source image data.

The responsibility for allocating the buffer for the transcoded data has been placed in the transcoder with the intent that some hardware manufacturers may find it useful to place the transcoded data directly into on-board memory on their video board. If the transcoding operation is being performed on a QuickTime movie, the transcoded data pointer will be almost immediately passed on to a decompressor. If the decompressor is implemented in hardware, some performance may be increased because the transcoded data is already loaded onto the decompression hardware.

ImageTranscoderDisposeData

Disposes of transcoded data.

```
pascal ComponentResult ImageTranscoderDisposeData (
                    ImageTranscoderComponent itc,
                    void *dstData);
```

`itc` The image transcoder component.

`dstData` Contains a pointer to the transcoded data.

DISCUSSION

When the client of the image transcoder component is done with a piece of transcoded data, `ImageTranscoderDisposeData` must be called with a pointer to the transcoded data. The image transcoder component should not make any assumptions about the maximum number of outstanding pieces of transcoded data or the order in which the transcoding data will be disposed.

ImageTranscoderEndSequence

Ends an image transcoding sequence.

```
pascal ComponentResult ImageTranscoderEndSequence
                    (ImageTranscoderComponent itc);
```

`itc` The image transcoder component whose transcoder sequence is ending.

DISCUSSION

`ImageTranscoderEndSequence` is called when there are no more frames of data to be transcoded using the parameters specified in the previous call to `ImageTranscoderBeginSequence`. After calling this function the component will either be closed or receive another call to `ImageTranscoderBeginSequence` with a different image description. (For example, the dimensions of the source image may be different.)

Movie Controller Components

Contents

This chapter discusses new features and changes to movie controller components as documented in Chapter 2 of *Inside Macintosh: QuickTime Components*.

New Features of Movie Controller Components

The new features of Movie Controller Components include five new actions and one new function. Additionally, one new flag has been defined to be returned by `MCGetControllerInfo` function.

Using Movie Controller Components

Changing the Shape of the Cursor

Many applications change the shape of the cursor depending on what it's currently over. The standard movie controller never changes the cursor, but other movie controllers may need to. An example of this is the QuickTime VR movie controller. Unfortunately, many applications need to control the cursor themselves—when a movie controller changes the cursor, these applications change it back immediately.

A simple solution is for applications to change the cursor only when it's first placed over a movie. (To determine whether a pointer is over the movie, use `mcPointInMovieController`.) After that, let the movie controller control the cursor until it exits the area over the movie. To give the movie controller the opportunity to change the cursor's shape, you must call either `MCIsPlayerEvent` or `MCIdle` while the cursor is over the movie, even if the movie is stopped. You can use the `mcActionSetCursorSettingEnabled` action to disable changes by movie controllers.

Movie Controller Components Reference

This section describes the new constants and functions associated with movie controller components.

The movie controller has always supported actions for setting the selection. With QuickTime 2.5, there are two new actions for getting the selection. In addition, QuickTime 2.1 added one new action that prerolls the movie before playing it, and two more actions that enable your application to control whether the movie controller can change the cursor.

Movie Controller Actions

This section discusses five new actions, which are integer constants (defined by the `mcAction` data type) used by movie controller components. Applications that use movie controller components can invoke these actions by calling the `MCDoAction` function.

This section does not describe all the existing constants documented in *Inside Macintosh: QuickTime Components*. Only the new constants are shown.

```
enum {
    mcActionGetSelectionBegin               = 53,   /* param is TimeRecord*/
    mcActionGetSelectionDuration            = 54,   /* param is TimeRecord*/
    mcActionPrerollAndPlay                  = 55,   /* param is Fixed, play rate*/
    mcActionGetCursorSettingEnabled         = 56,   /* param is pointer to Boolean*/
    mcActionSetCursorSettingEnabled         = 57,   /* param is Boolean*/
    mcActionSetColorTable                   = 58,   /* param is CTabHandle*/
};
typedef short mcAction;
```

Actions for Use by Applications

`mcActionGetSelectionBegin`

The parameter must contain a pointer to a time structure. The time returned is in the time scale of the movie. The returned time indicates the start time of the current user time selection.

`mcActionGetSelectionDuration`

The parameter must contain a pointer to a time structure. The time value returned is in the time scale of the movie. The returned time indicates the duration of the current user time selection. If there is no selection, this value will be zero.

`mcActionPrerollAndPlay`

Your application can use this action to preroll a movie and then immediately play it. You should use this action whenever a movie controller is used and the movie needs to be played programmatically.

The parameter data must contain a fixed value that indicates the rate of play. Values greater than 0 correspond to forward rates; values less than 0 play the movie backward. A value of 0 stops the movie.

`mcActionGetCursorSettingEnabled`

Your application can use this action to determine whether cursor switching is enabled for a movie controller.

The parameter data must contain a pointer to a Boolean value—a value of `true` indicates that cursor switching is enabled. By default, this value is set to `true`.

`mcActionSetCursorSettingEnabled`

Your application can use this action to control whether the movie controller can change the cursor.

The parameter data must contain a Boolean value. Set this value to true to enable the movie controller to change the cursor. Set it to false to disable cursor switching.

Some movie controllers (QuickTime VR, for example) change the cursor while the pointer is over the movie to indicate that the pointer is over a hot spot. If you do not want the movie controller to change the cursor, you should use this action to prevent the movie controller from changing the cursor.

`mcActionSetColorTable`

Your application can use this action to determine when the movie controller is going to set a new color table. Setting a color table causes the window's palette to be updated to the new color table. Applications can use this action to

monitor or control the movie controller's current color environment.

Movie Controller Functions

This section describes one new function that is supported by movie controller components that handle movie events. A new flag for an existing function has also been added.

Handling Movie Events

QuickTime 2.1 provides two changes to handling movie events as documented in Chapter 2 of *Inside Macintosh: QuickTime Components*.. First, a new flag can now be returned by the `MCGetControllerInfo` function. This flag indicates when a movie is interactive, and therefore does not make sense to play. Second, while it has it has always been possible to determine if a point is contained in a movie (using `PtInMovie`), the new `MCPtInController` function provides a way to determine if a point is in the controls of a movie.

MCGetControllerInfo

The `MCGetControllerInfo` function returns a new flag to indicate that the movie is interactive and, therefore, cannot be played from start to end. For example, because users interact with a QuickTime VR movie, it cannot be played.

The `someflags` parameter to the `MCGetControllerInfo` function may return the following additional flag:

```
enum {
    mcInfoMovieIsInteractive    = 1 << 10,
};
```

Flag description

`mcInfoMovieIsInteractive`

If this flag is set to 1, the movie is interactive.

MCPtInController

Reports whether a point is in the control area of a movie.

```
pascal ComponentResult MCPtInController (
                    MovieController mc,
                    Point thePt,
                    Boolean *inController);
```

`mc` Specifies the movie controller for the operation. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function, or from the `NewMovieController` function.

`thePt` Specifies the point to be checked. This point must be passed in local coordinates to the controller's window. This point is checked only against the Movie Controller's controls, not the movie itself.

`inController` Returns `true` if the point is in the controller; `false` if it is not.

DISCUSSION

While you could always determine if a point is contained in a movie (using `PtInMovie`), the `MCPtInController` function allows you to determine if a point is in the control area of a movie.

Sequence Grabber Components

Contents

This chapter discusses new features and changes to sequence grabber components as documented in Chapter 5 of *Inside Macintosh: QuickTime Components*.

New Features of Sequence Grabber Components

Sequence grabber components now allow you to assign a specific file to each channel. This allows you to collect data into more than one file at a time, which can result in improved performance by defining the files for different channels on different devices. These destination containers are referred to as *sequence grabber outputs.* See "Working with Sequence Grabber Outputs" (page 7-13) for a complete discussion.

Sequence grabber components now use data handler components when writing movie data. This provides greater flexibility, especially when working with special storage devices (such as networks).

As discussed in Chapter 1, "Movie Toolbox," QuickTime 2.0 introduced timecode tracks to QuickTime movies. The sequence grabber automatically creates a timecode track if the video digitizer component contains timecode information. In order to support timecode tracks, the sequence grabber also provides two functions that let you identify the source information associated with video data that contains timecode information. For more information about timecodes and the timecode media handler, see Chapter 1, "Movie Toolbox."

Improved Support for Digitizing Video in Windows

Prior to QuickTime 2.1, when displaying the output of a sequence grabber video channel in a window, an application would have to pause the sequence grabber before moving or resizing a window, before a menu was pulled down, or whenever the application was put into the background. If an application failed to take these precautions, it was possible for the digitized video to draw outside of the window with which it was associated.

QuickTime 2.1 solves these problems so that an application using a sequence grabber video channel in a window no longer has to take any special precautions to ensure that the video remains within the window. As long as the

application calls `SGIdle` regularly, the sequence grabber will automatically take care of pausing and unpausing the video as necessary.

Sequence Grabber Components Reference

This section describes the new constants and functions associated with sequence grabber components.

Constants

This section describes the new constants for sequence grabber components.

Flags

QuickTime 1.6.1 added a new flag to the `grabPictCurrentImage` parameter to the `SGGrabPict` function.

```
enum {
    grabPictCurrentImage            = 4
};
```

Constant description

`grabPictCurrentImage`
Set this flag to 1 to provide the fastest possible image capture. Although this flag may fail under certain circumstances, this failure is recoverable; it just will not return a picture. You can then call `SGGrabPict` again without the flag set. This routine does not pause the current preview or grab the next frame. It just causes the currently displayed image to be captured. It's a good idea to call `SGPause` before calling `SGGrabPict` with this flag.

The `flags` parameter to the `SGSettingsDialog` function is a reserved flag and can only be set to 0. QuickTime 2.1 provides a new flag value you can use to indicate that you want to display only panels that make sense for previewing.

```
enum {
    seqGrabSettingsPreviewOnly      = 1
};
```

Constant description

`seqGrabSettingsPreviewOnly`
Set this flag to indicate that the user will be using the dialog provided by `SGSettingsDialog` to configure the Sequence Grabber for previewing only, not for recording. The `SGSettingsDialog` will automatically exclude any panels which aren't necessary for configuring for previewing, such as video or audio compression settings. Otherwise, set the `flags` parameter to 0.

Sequence Grabber Component Functions

This section describes the new and changed functions provided by sequence grabber components.

Configuring Sequence Grabber Components

SGSetDataRef

Specifies the destination data reference for a record operation.

```
pascal ComponentResult SGSetDataRef (
                    SeqGrabComponent s,
                    Handle dataRef,
                    OSType dataRefType,
                    long whereFlags);
```

`s`
Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`dataRef` Contains a handle to the information that identifies the destination container.

`dataRefType` Specifies the type of data reference. If the data reference is an alias, you must set the parameter to `rAliasType` (`'alis'`), indicating that the reference is an alias.

`whereFlags` Contains flags that control the record operation. You must set either the `seqGrabToDisk` flag or the `seqGrabToMemory` flag to 1 (set unused flags to 0):

`seqGrabToDisk`
Instructs the sequence grabber component to write the recorded data to a QuickTime movie in the container specified by the `dataRef` parameter. If you set this flag to 1, the sequence grabber writes the data to the container as the data is recorded. Set this flag to 0 if you set the `seqGrabToMemory` flag to 1 (only one of these two flags may be set to 1).

`seqGrabToMemory`
Instructs the sequence grabber component to store the recorded data in memory until the recording process is complete. The sequence grabber then writes the recorded data to the container specified by the `dataRef` parameter. This technique provides better performance than recording directly to the container, but limits the amount of data you can record. Set this flag to 1 to record to memory. Set this flag to 0 if you set the `seqGrabToDisk` flag to 1 (only one of these two flags may be set to 1).

`seqGrabDontUseTempMemory`
Prevents the sequence grabber component from using temporary memory during the record operation. By default, the sequence grabber component and its channel components use as much temporary memory as necessary to perform the record operation. Set this flag to 1 to

prevent the sequence grabber component and its channel components from using temporary memory.

`seqGrabAppendToFile`

Directs the sequence grabber component to add the recorded data to the data fork of the container specified by the `dataRef` parameter. By default, the sequence grabber component deletes the container and creates a new file containing one movie and the corresponding movie resource. Set this flag to 1 to cause the sequence grabber component to append the recorded data to the data fork of the container and create a new movie resource in that file.

`seqGrabDontAddMovieResource`

Prevents the sequence grabber component from adding the new movie resource to the container specified by the `dataRef` parameter. By default, the sequence grabber component creates a new movie resource and adds that resource to the container. Set this flag to 1 to prevent the sequence grabber component from adding the movie resource to the container. You are then responsible for adding the resource to a file, if you so desire.

`seqGrabDontMakeMovie`

Prevents the sequence grabber component from creating a movie. By default, the sequence grabber component creates a new movie resource and adds the captured data to that movie. If you set this flag to 1, the sequence grabber still calls your data function, but does not write any data to the movie file.

`seqGrabDataProcIsInterruptSafe`

Specifies that your data function is interrupt-safe, and may be called at interrupt time. This allows the sequence grabber component to present the captured data as soon as possible. Note that not all sequence grabber

channel components may use this feature. It is currently supported only by sequence grabber sound channels.

DISCUSSION

The `SGSetDataRef` function allows you to specify the destination for a record operation using a data reference, and to specify other options that govern the operation. This function is similar to the `SGSetDataOutput` function, and provides you an alternative way to specify the destination.

If you are performing a preview operation, you do not need to use the `SGSetDataRef` function.

RESULT CODES

`notEnoughMemoryToGrab`	–9403	Insufficient memory for operation
`notEnoughDiskSpaceToGrab`	–9404	Insufficient disk space for operation

File Manager errors
Memory Manager errors

SGGetDataRef

The `SGGetDataRef` function allows you to determine the data reference that is currently assigned to a sequence grabber component and the control flags that would govern a record operation.

```
pascal ComponentResult SGGetDataRef (
                    SeqGrabComponent s,
                    Handle *dataRef,
                    OSType *dataRefType,
                    long *whereFlags);
```

`s` Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`dataRef` Contains a pointer to a handle that is to receive the information that identifies the destination container.

`dataRefType` Specifies a pointer to a field that is to receive the type of data reference.

`whereFlags` Contains a pointer to a long integer that is to receive flags that control the record operation. The following flags are defined (unused flags are set to 0):

`seqGrabToDisk`
Instructs the sequence grabber component to write the recorded data to a QuickTime movie in the container specified by the `dataRef` parameter. If this flag is set to 1, the sequence grabber writes the data to the container as the data is recorded.

`seqGrabToMemory`
Instructs the sequence grabber component to store the recorded data in memory until the recording process is complete. The sequence grabber then writes the recorded data to the container specified by the `dataRef` parameter. This technique provides better performance than recording directly to the movie file, but limits the amount of data you can record. If this flag is set to 1, the sequence grabber component is recording to memory.

`seqGrabDontUseTempMemory`
Prevents the sequence grabber component from using temporary memory during the record operation. By default, the sequence grabber component and its channel components use as much temporary memory as necessary to perform the record operation. If this flag is set to 1, the sequence grabber component and its channel components do not use temporary memory.

`seqGrabAppendToFile`
Directs the sequence grabber component to add the recorded data to the data fork of the

container specified by the `dataRef` parameter. By default, the sequence grabber component deletes the container and creates a new file containing one movie and its movie resource. If this flag is set to 1, the sequence grabber component appends the recorded data to the data fork of the container and creates a new movie resource in that file.

`seqGrabDontAddMovieResource`

Prevents the sequence grabber component from adding the new movie resource to the container specified by the `dataRef` parameter. By default, the sequence grabber component creates a new movie resource and adds that resource to the container. If this flag is set to 1, the sequence grabber component does not add the movie resource to the container. You are then responsible for adding the resource to a file, if you so desire.

`seqGrabDontMakeMovie`

Prevents the sequence grabber component from creating a movie. By default, the sequence grabber component creates a new movie resource and adds the captured data to that movie. If this flag is set to 1, the sequence grabber still calls your data function, but does not write any data to the container.

`seqGrabDataProcIsInterruptSafe`

Specifies that your data function is interrupt-safe, and may be called at interrupt time. This allows the sequence grabber component to present the captured data as soon as possible. Note that not all sequence grabber channel components may use this feature.

DISCUSSION

The `SGGetDataRef` function allows you to determine the data reference that is currently assigned to a sequence grabber component and the control flags that would govern a record operation.

You set these characteristics by calling the `SGSetDataRef` function, which is described in the previous section. If you have not set these characteristics before calling the `SGGetDataRef` function, the returned data is meaningless.

RESULT CODES

Memory Manager errors

SGSettingsDialog

The `SGSettingsDialog` function has a new flag value that you can pass in the `flags` parameter. Previously, you could only set this parameter to 0. Pass the new flag, `seqGrabSettingsPreviewOnly`, to indicate that you want to display only panels that make sense for previewing. In particular, the video compression will not be displayed. Use this flag for applications that allow a live video signal to be viewed but not captured.

`flags` Either set this to 0 or to `seqGrabSettingsPreviewOnly`. The function supports the following flag value:

`seqGrabSettingsPreviewOnly`
Use this value if you want to view but not capture a live video signal. Otherwise, set the `flags` parameter to 0.

Controlling Sequence Grabber Components

SGGrabPict

QuickTime 1.6.1 added a new flag to the `grabPictCurrentImage` parameter to the `SGGrabPict` function.

```
enum {
    grabPictCurrentImage            = 4
};
```

Constant descriptions

`grabPictCurrentImage`
: Set this flag to 1 to provide the fastest possible image capture. Although this flag may fail under certain circumstances, this failure is recoverable; it just will not return a picture. You can then call `SGGrabPict` again without the flag set. This routine does not pause the current preview or grab the next frame. It just causes the currently displayed image to be captured. It's a good idea to call `SGPause` before calling `SGGrabPict` with this flag.

SGGetMode

Returns the mode for a sequence grabber component.

```
pascal ComponentResult SGGetMode (
                    SeqGrabComponent s,
                    Boolean *previewMode,
                    Boolean *recordMode);
```

`s`
: Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`previewMode`
: Contains a pointer to a Boolean. The sequence grabber component sets this field to `true` if the component is in preview mode.

`recordMode`
: Contains a pointer to a Boolean. The sequence grabber component sets this field to `true` if the component is in record mode.

DISCUSSION

The `SGGetMode` function provides a convenient mechanism for determining whether a sequence grabber component is in preview mode or record mode.

Working with Sequence Grabber Outputs

In order to allow sequence grabber components to capture to more than one data reference at a time, QuickTime 2.0 introduced the concept of a sequence grabber output. A *sequence grabber output* ties a sequence grabber channel to a specified data reference for output of captured data

If you are capturing to a single movie file, you can continue to use the `SGSetDataOutput` function (or the new `SGSetDataRef` function) to specify the sequence grabber's destination. However, if you want to capture movie data into several different files or data references, you must use sequence grabber outputs to do so. Even if you are using outputs, you must still use the `SGSetDataOutput` function or the `SGSetDataRef` function to identify where the sequence grabber should create the movie resource.

You are responsible for creating outputs, assigning them to sequence grabber channels, and disposing of them when you are done. Sequence grabber components provide a number of functions for managing outputs: the `SGNewOutput` function creates a new output; the `SGDisposeOutput` function disposes of an output; the `SGSetOutputFlags` function configures the output; the `SGSetChannelOutput` function assigns an output to a channel; and the `SGGetDataOutputStorageSpaceRemaining` function determines how much space is left in the output.

SGNewOutput

Creates a new sequence grabber output.

```
pascal ComponentResult SGNewOutput (
                    SeqGrabComponent s,
                    Handle dataRef,
                    OSType dataRefType,
                    long whereFlags,
                    SGOutput *output);
```

`s` Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`dataRef` Contains a handle to the information that identifies the destination container.

`dataRefType` Specifies the type of data reference. If the data reference is an alias, you must set the parameter to `rAliasType` (`'alis'`), indicating that the reference is an alias.

`whereFlags` Contains flags that control the record operation. You must set either the `seqGrabToDisk` flag or the `seqGrabToMemory` flag to 1 (set unused flags to 0):

`seqGrabToDisk`
Instructs the sequence grabber component to write the recorded data to a QuickTime movie in the container specified by the `dataRef` parameter. If you set this flag to 1, the sequence grabber writes the data to the container as the data is recorded. Set this flag to 0 if you set the `seqGrabToMemory` flag to 1 (only one of these two flags may be set to 1).

`seqGrabToMemory`
Instructs the sequence grabber component to store the recorded data in memory until the recording process is complete. The sequence grabber then writes the recorded data to the container specified by the `dataRef` parameter. This technique provides better performance than recording directly to the container, but limits the amount of data you can record. Set this flag to 1 to record to memory. Set this flag to 0 if you set the `seqGrabToDisk` flag to 1 (only one of these two flags may be set to 1).

`seqGrabDontUseTempMemory`
Prevents the sequence grabber component from using temporary memory during the record operation. By default, the sequence grabber component and its channel components use as

much temporary memory as necessary to perform the record operation. Set this flag to 1 to prevent the sequence grabber component and its channel components from using temporary memory.

`seqGrabAppendToFile`

Directs the sequence grabber component to add the recorded data to the data fork of the container specified by the `dataRef` parameter. By default, the sequence grabber component deletes the container and creates a new file containing one movie and the corresponding movie resource. Set this flag to 1 to cause the sequence grabber component to append the recorded data to the data fork of the container and create a new movie resource in that file.

`seqGrabDontAddMovieResource`

Prevents the sequence grabber component from adding the new movie resource to the container specified by the `dataRef` parameter. By default, the sequence grabber component creates a new movie resource and adds that resource to the container. Set this flag to 1 to prevent the sequence grabber component from adding the movie resource to the container. You are then responsible for adding the resource to a file, if you so desire.

`seqGrabDontMakeMovie`

Prevents the sequence grabber component from creating a movie. By default, the sequence grabber component creates a new movie resource and adds the captured data to that movie. If you set this flag to 1, the sequence grabber still calls your data function, but does not write any data to the movie file.

`seqGrabDataProcIsInterruptSafe`

Specifies that your data function is interrupt-safe, and may be called at interrupt time. This allows the sequence grabber

component to present the captured data as soon as possible. Note that not all sequence grabber channel components may use this feature.

`output` Contains a pointer to a sequence grabber output. The sequence grabber component returns an output identifier. You can then use this identifier with other sequence grabber component functions.

DISCUSSION

The `SGNewOutput` function creates a new sequence grabber output. You specify the output's destination container using a data reference. Once you have created the sequence grabber output, you can use the `SGSetChannelOutput` function to assign the output to a sequence grabber channel.

RESULT CODES

`paramErr`	–50	Invalid parameter specified

File Manager errors
Memory Manager errors

SGDisposeOutput

Disposes of an existing output.

```
pascal ComponentResult SGDisposeOutput (
                    SeqGrabComponent s,
                    SGOutput output);
```

`s` Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`output` Identifies the sequence grabber output for this operation. You obtain this identifier by calling the `SGNewOutput` function.

DISCUSSION

You use the `SGDisposeOutput` function to dispose of an existing output. If any sequence grabber channels are using this output, the sequence grabber component assigns them to an undefined output.

Note that you cannot dispose of an output when the sequence grabber component is in record mode.

RESULT CODES

`cantDoThatInCurrentMode`	–9402	Request invalid in current mode

SGSetChannelOutput

Assigns an output to a channel.

```
pascal ComponentResult SGSetChannelOutput (
                    SeqGrabComponent s,
                    SGChannel c,
                    SGOutput output);
```

`s` Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`c` Identifies the channel for this operation. Provide your connection identifier. You connect to a channel component by calling the `SGNewChannel` or `SGNewChannelFromComponent` functions.

`output` Identifies the sequence grabber output for this operation. You obtain this identifier by calling the `SGNewOutput` function.

DISCUSSION

You use the `SGSetChannelOutput` function to assign an output to a channel. Note that when you call the `SGSetDataRef` or `SGSetDataOutput` functions the sequence grabber component sets every channel to the specified file or container. If you

want to use different outputs, you must use this function to assign the channels appropriately. One output may be assigned to one or more channels.

RESULT CODES

badSGChannel	–9406	Invalid channel specified

SGSetOutputFlags

Configures an existing sequence grabber output.

```
pascal ComponentResult SGSetOutputFlags (
                    SeqGrabComponent s,
                    SGOutput output,
                    long whereFlags);
```

`s` Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`output` Identifies the sequence grabber output for this operation. You obtain this identifier by calling the `SGNewOutput` function.

`whereFlags` Contains flags that control the record operation. You must set either the `seqGrabToDisk` flag or the `seqGrabToMemory` flag to 1 (set unused flags to 0):

`seqGrabToDisk`
Instructs the sequence grabber component to write the recorded data to a QuickTime movie in the container specified by the `dataRef` parameter. If you set this flag to 1, the sequence grabber writes the data to the container as the data is recorded. Set this flag to 0 if you set the `seqGrabToMemory` flag to 1 (only one of these two flags may be set to 1).

`seqGrabToMemory`

Instructs the sequence grabber component to store the recorded data in memory until the recording process is complete. The sequence grabber then writes the recorded data to the container specified by the `dataRef` parameter. This technique provides better performance than recording directly to the container, but limits the amount of data you can record. Set this flag to 1 to record to memory. Set this flag to 0 if you set the `seqGrabToDisk` flag to 1 (only one of these two flags may be set to 1).

`seqGrabDontUseTempMemory`

Prevents the sequence grabber component from using temporary memory during the record operation. By default, the sequence grabber component and its channel components use as much temporary memory as necessary to perform the record operation. Set this flag to 1 to prevent the sequence grabber component and its channel components from using temporary memory.

`seqGrabAppendToFile`

Directs the sequence grabber component to add the recorded data to the data fork of the container specified by the `dataRef` parameter. By default, the sequence grabber component deletes the container and creates a new file containing one movie and the corresponding movie resource. Set this flag to 1 to cause the sequence grabber component to append the recorded data to the data fork of the container and create a new movie resource in that file.

`seqGrabDontAddMovieResource`

Prevents the sequence grabber component from adding the new movie resource to the container specified by the `dataRef` parameter. By default, the sequence grabber component creates a new movie resource and adds that resource to the

container. Set this flag to 1 to prevent the sequence grabber component from adding the movie resource to the container. You are then responsible for adding the resource to a file, if you so desire.

`seqGrabDontMakeMovie`

Prevents the sequence grabber component from creating a movie. By default, the sequence grabber component creates a new movie resource and adds the captured data to that movie. If you set this flag to 1, the sequence grabber still calls your data function, but does not write any data to the movie file.

`seqGrabDataProcIsInterruptSafe`

Specifies that your data function is interrupt-safe, and may be called at interrupt time. This allows the sequence grabber component to present the captured data as soon as possible. Note that not all sequence grabber channel components may use this feature.

DISCUSSION

The `SGSetOutputFlags` function allows you to configure an existing sequence grabber output.

RESULT CODES

paramErr	–50	Invalid parameter specified
cantDoThatInCurrentMode	–9402	Request invalid in current mode

SGGetDataOutputStorageSpaceRemaining

Returns the amount of space remaining in the data reference associated with an output.

```
pascal ComponentResult SGGetDataOutputStorageSpaceRemaining (
                    SeqGrabComponent s,
                    SGOutput output,
                    unsigned long *space);
```

s	Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's OpenDefaultComponent or OpenComponent function.
output	Identifies the sequence grabber output for this operation. You obtain this identifier by calling the SGNewOutput function.
space	Contains a pointer to an unsigned long. The sequence grabber component returns a value that indicates the number of bytes of space remaining in the data reference associated with the output.

DISCUSSION

The SGGetDataOutputStorageSpaceRemaining function allows you to determine the amount of space remaining in the data reference associated with an output. Use this function in place of the SGGetStorageSpaceRemaining function in cases where you are working with more than one output.

RESULT CODES

paramErr	–50	Invalid parameter specified

Sequence Grabber Channel Components

Contents

This chapter discusses new features and changes to sequence grabber channel components as documented in Chapter 6 of *Inside Macintosh: QuickTime Components*.

New Features of Sequence Grabber Channel Components

Support for Sound Data Compression

The sound sequence grabber channel now supports compression of sound data in software when the requested format is not supported directly by the sound input driver. This feature is available only when Sound Manager 3.1 or later is installed. You can now directly capture sound in IMA and µLaw formats. Because new audio compressors and decompressor can be installed by system extensions (such as QuickTime Conferencing), other audio compression formats may also be available.

Support for Sound Capture at Any Sample Rate

QuickTime 2.5 enhanced the sequence grabber sound channel to allow sound to be captured at any sample rate. The sample rate is specified, as in the past, by using `SGSetSoundInputRate`. If the requested rate is not one of the hardware rates, the sound will be captured using the closest available hardware sample rate and will be rate converted in software to the requested rate.

*Working With Channel Characteristics

The sequence grabber supports two new functions, `SGChannelSetDataSourceName` and `SGChannelGetDataSourceName`, that allow you to specify the source identification information associated with a sequence grabber channel. For more information about timecodes and the timecode media handler, see Chapter 1, “Movie Toolbox.”

Sequence Grabber Channel Components Reference

Functions

This section describes the new functions specific to the Apple-supplied sequence grabber channel component.

Configuration Functions for All Channel Components

SGChannelSetRequestedDataRate

Specifies the maximum requested data rate for a channel.

```
pascal ComponentResult SGChannelSetRequestedDataRate (
                    SGChannel c,
                    long bytesPerSecond);
```

`c` Identifies the channel connection for this operation.

`bytesPerSecond`
Specifies the maximum data rate requested by the sequence grabber component. The sequence grabber component sets this parameter to 0 to remove any data-rate restrictions.

DISCUSSION

The `SGChannelSetRequestedDataRate` function allows the sequence grabber component to specify the maximum rate at which it would like to receive data from your channel component.

The data rate supplied by the sequence grabber component represents a requested data rate. Your component may not be able to observe that rate under all conditions.

RESULT CODES

badComponentSelector	0x80008002	Function not supported

SGChannelGetRequestedDataRate

Returns the current maximum data rate requested for a channel.

```
pascal ComponentResult SGChannelGetRequestedDataRate (
                    SGChannel c,
                    long *bytesPerSecond);
```

c — Identifies the channel connection for this operation.

bytesPerSecond — Points to a field that is to receive the maximum data rate requested by the sequence grabber component.This field is set to 0 if the sequence grabber has not set any restrictions.

DESCRIPTION

The SGChannelGetRequestedDataRate function allows the sequence grabber component to retrieve the current maximum data rate value from your channel component.

RESULT CODES

badComponentSelector	0x80008002	Function not supported

SGSetPreferredPacketSize

Sets the preferred packet size for the sequence grabber channel component.

```
pascal ComponentResult SGSetPreferredPacketSize (
                    SGChannel c,
                    long preferredPacketSizeInBytes);
```

`c` Identifies the channel connection for this operation.

`preferredPacketSizeInBytes`
The preferred packet size in bytes.

DESCRIPTION

This function was added in QuickTime 2.5 to support video conferencing applications.

SGGetPreferredPacketSize

Returns the preferred packet size for the sequence grabber component.

```
pascal ComponentResult SGGetPreferredPacketSize (
                    SGChannel c,
                    long *preferredPacketSizeInBytes);
```

`c` Identifies the channel connection for this operation.

`preferredPacketSizeInBytes`
The preferred packet size in bytes.

DESCRIPTION

This function was added in QuickTime 2.5 to support video conferencing applications.

SGChannelSetDataSourceName

Sets the data source name for a track.

```
pascal ComponentResult SGChannelSetDataSourceName (
                    SGChannel c,
                    const Str255 name,
                    ScriptCode scriptTag);
```

c	Identifies the channel connection for this operation.
name	Identifies a string that contains the source identification information.
scriptTag	Specifies the language of the source identification information.

DISCUSSION

The SGChannelSetDataSourceName function allows you to set the source informationfor a sequence grabber channel. You must set this information before you start digitizing.

This source information identifies the source of the video data (say, a videotape name). The sequence grabber channel stores this information in a timecode track in the movie created after the capture is complete. If the video digitizer does not provide timecode information, the sequence grabber does not save this information.

This function is currently supported only by video channels.

SGChannelGetDataSourceName

Returns the data source name for a track.

```
pascal ComponentResult SGChannelGetDataSourceName (
                    SGChannel c,
                    Str255 name,
                    ScriptCode *scriptTag);
```

c	Identifies the channel connection for this operation.
name	Identifies a string that is to receive the source identification information. Set this parameter to nil if you do not want to retrieve the name.
scriptTag	Specifies a field that is to receive the source information's language code. Set this parameter to nil if you do not want this information.

DESCRIPTION

The `SGChannelGetDataSourceName` function allows you to get the source information specified with `SGChannelSetDataSourceName`.

SGSetAdditonalSoundRates

Allows an application to specify a list of sound sample rates to be included in the Sequence Grabber's Sound settings dialog. If any of requested rates are not supported directly by the available sound capture hardware, sound will be captured at one of the available hardware rates and then rate converted in software to the requested rate.

```
pascal ComponentResult SGSetAdditionalSoundRates(SGChannel c,
                    Handle rates)
```

`c` Identifies the channel connection for this operation.

`rates` A handle containing a list of unsigned 32 bit fixed point values. The sequence grabber channel determines the number of sample rates contained in the handle based on the size of the handle.

DISCUSSION

The sequence grabber channel makes a copy of the additional rates handle. Therefore, your application can immediately dispose of the additional rates handle after making this call.

SGGetAdditonalSoundRates

Returns the additional sound sample rates added to the specified sequence grabber sound channel.

```
pascal ComponentResult SGGetAdditionalSoundRates(SGChannel c,
                    Handle *rates)
```

c	Identifies the channel connection for this operation.
rates	A pointer to handle where the list of additional sample rates should be returned.

DISCUSSION

SGGetAdditionalSoundRates returns a copy of the list of additional samples rates passed to the SSGetAdditionalSoundRates previously. If no additional sample rates have been set, SGGetAdditionalSoundRates sets the rates handle to nil. The caller of this routine is responsible for disposing of the returned rates handle.

SGSetUserVideoCompressorList

Allows an application to specify the list of video compression formats to be included in the Sequence Grabber's Video settings dialog. This allows an application to limit the number of video compression formats that will be displayed to the user. For applications using the sequence grabber for a very specific purpose, this allows inappropriate compression choices to be filtered out.

```
pascal ComponentResult SGSetUserVideoCompressorList(SGChannel c,
                    Handle compressorTypes)
```

c	Identifies the channel connection for this operation.
compressorTypes	A handle containing a list of OSTypes indicating which video compression formats should be displayed. The sequence grabber channel determines the number of video compression formats contained in the handle based on the size of the handle.

DISCUSSION

The sequence grabber channel makes a copy of the video compression formats handle. Therefore, your application can immediately dispose of the video compression formats handle after making this call.

SGGetUserVideoCompressorList

Returns the video compression formats to be displayed by the specified sequence grabber video channel.

```
pascal ComponentResult SGGetUserVideoCompressorList(SGChannel c,
                    Handle *compressorTypes)
```

`c` Identifies the channel connection for this operation.

`compressorTypes`
A pointer to handle where the list of video compression formats should be returned.

DISCUSSION

`SGGetUserVideoCompressorList` returns a copy of the list of video compression formats passed to the `SGSetUserVideoCompressorList` previously. If no video compression formats have been set, `SGGetUserVideoCompressorList` sets the compressorTypes handle to `nil`. The caller of this routine is responsible for disposing of the returned video compression formats handle.

Video Digitizer Components

Contents

This chapter discusses changes to video digitizer components as documented in Chapter 8 of *Inside Macintosh: QuickTime Components*. This chapter describes the new and changed constants, data types, and functions provided by these components.

New Features of Video Digitizer Components

As discussed in Chapter 1, "Movie Toolbox," QuickTime 2.0 introduced timecode tracks to QuickTime movies. Video digitizers may return timecode information for an incoming video signal by responding to the new `VDGetTimeCode` function described in this chapter. For more information about timecodes and the timecode media handler, see Chapter 1, "Movie Toolbox."

Video Digitizer Components Reference

Constants

Input Formats

You use the `VDGetInputFormat` function to find out the video format employed by a specified input. QuickTime defines one new constant that you can use for video digitizers that support a tuner input.

```
enum {
    tvTunerIn           = 6
};
```

Constant description

`tvTunerIn`
: The input video signal is from the television tuner connection.

Video Digitizer Component Functions

Controlling Compressed Source Devices

In QuickTime 1.5, video digitizers could provide compressed data directly to clients. However, there was no way to preflight the settings for compression. In QuickTime 2.1, a new function, `VDGetCompressionTime`, has been added to allow the video digitizer to quantize requested quality values to the actual quality levels that will be used.

VDGetCompressionTime

Your component receives the `VDGetCompressionTime` request whenever a client of the digitizer wants to confirm or quantize its compression settings.

```
pascal VideoDigitizerError VDGetCompressionTime (
                    VideoDigitizerComponent ci,
                    OSType compressionType,
                    short depth,
                    Rect *srcRect,
                    CodecQ *spatialQuality,
                    CodecQ *temporalQuality,
                    unsigned long *compressTime);
```

`ci` Specifies the video digitizer component for the request. Applications obtain this reference from the Component Manager's `OpenComponent` function.

`compressionType`
Specifies a compressor type. This value corresponds to the component subtype of the compressor component. See the chapter "Image Compression Manager" in *Inside Macintosh: QuickTime* for more information about compressor types and for valid values for this parameter.

`depth` Specifies the depth at which the image is to be compressed. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 33, 34, 36, and 40 indicate 1-bit, 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images.

`srcRect` Contains a pointer to a rectangle defining the portion of the source image to compress.

`spatialQuality`
Contains a pointer to a field containing the desired compressed image quality. The compressor sets this field to the closest actual quality that it can achieve. See the chapter "Image Compression Manager" in *Inside Macintosh: QuickTime* for valid values. A value of `nil` indicates that the client does not want this information.

`temporalQuality`
Contains a pointer to a field containing the desired sequence temporal quality. The compressor sets this field to the closest actual quality that it can achieve. See the chapter "Image Compression Manager" in *Inside Macintosh: QuickTime* for valid values. A value of `nil` indicates that the client does not want this information.

`compressTime` Contains a pointer to a field to receive the compression time, in milliseconds. If your component cannot determine the amount of time required to compress the image, set this field to 0. A value of `nil` indicates that the client does not want this information.

DISCUSSION

The Sequence Grabber's video compression settings dialog uses this function to snap the quality slider to the correct value when working with a compression type that is provided by the video digitizer.

Your component returns a long integer indicating the maximum number of milliseconds it would require to compress the specified image.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified

Controlling Digitization

This section describes one new video digitizer component function, VDSetDataRate, that instructs your video digitizer component to observe a specified rate of data delivery.

VDSetDataRate

The VDSetDataRate function instructs your video digitizer component to limit the rate at which it delivers compressed, digitized video data.

```
pascal VideoDigitizerError VDSetDataRate (
                    VideoDigitizerComponent ci,
                    long bytesPerSecond);
```

ci — Specifies the video digitizer component for the request. Applications obtain this reference from the Component Manager's OpenComponent function.

bytesPerSecond — Specifies the maximum data rate requested by the application. This parameter is set to 0 to remove any data-rate restrictions.

DISCUSSION

This function is valid only for video digitizer components that can deliver compressed video (that is, components that support the VDCompressOneFrameAsync function). Components that support data-rate limiting set the codecInfoDoesRateConstrain flag to 1 in the compressFlags field of the VDCompressionList structure returned by the component in response to the VDGetCompressionTypes function.

Your video digitizer component should return this data-rate limit in the bytesPerSecond parameter of the existing VDGetDataRate function.

Controlling Packet Size

VDSetPreferredPacketSize

Sets the preferred packet size for digitizing.

```
pascal VideoDigitizerError VDSetPreferredPacketSize(
                    VideoDigitizerComponent ci,
                    long preferredPacketSizeInBytes);
```

`ci` Specifies the video digitizer component for the request. Applications obtain this reference from the Component Manager's `OpenComponent` function.

`preferredPacketSizeInBytes`
The preferred packet size in bytes.

DESCRIPTION

This function was added in QuickTime 2.5 to support video conferencing applications.

Utility Functions

This section describes two new utility functions that may be supported by some video digitizer components.

The `VDGetTimeCode` function allows an application to retrieve timecode information.

The `VDGetSoundInputSource` function allows an application to retrieve information about a digitizer's sound input source.

VDGetTimeCode

The `VDGetTimeCode` function instructs your video digitizer component to return timecode information for the incoming video signal.

```
pascal VideoDigitizerError VDGetTimeCode (
                    VideoDigitizerComponent ci,
                    TimeRecord *atTime,
                    TimeCodeDef *timeCodeFormat,
                    TimecodeTime *timeCodeTime);
```

`ci` Specifies the video digitizer component for the request. Applications obtain this reference from the Component Manager's `OpenComponent` function.

`atTime` Specifies a location to receive the QuickTime movie time value corresponding to the timecode information.

`timeCodeFormat`
Contains a pointer to a timecode definition structure. Your video digitizer component returns the movie's timecode definition information.

`timeCodeTime` Contains a pointer to a timecode record structure. Your video digitizer component returns the time value corresponding to the movie time contained in the `atTime` parameter.

DISCUSSION

Typically, this function is called once, at the beginning of a capture session. The use of `VDGetTimeCode` assumes that the time code for the entire capture session will be continuous.

For more information about the timecode data structures, see Chapter 1, "Movie Toolbox."

VDGetSoundInputSource

The `VDGetSoundInputSource` function instructs your video digitizer component to return the sound input source associated with a particular video input.

```
pascal VideoDigitizerError VDGetSoundInputSource (
                    VideoDigitizerComponent ci,
                    long videoInput,
                    long *soundInput);
```

`ci` Specifies the video digitizer component for the request. Applications obtain this reference from the Component Manager's `OpenComponent` function.

`videoInput` Specifies the input video source for this request. Video digitizer components number video sources sequentially, starting at 0. So, to request information about the first video source, an application sets this parameter to 0. Applications can get the number of video sources supported by a video digitizer component by calling the `VDGetNumberOfInputs` function.

`soundInput` The sound input index to use with the sound input driver returned by `VDGetSoundInputDriver`.

DISCUSSION

Some video digitizers may associate different sound inputs with each video input. The `VDGetSoundInputDriver` function returns the name of the sound input driver that the sound input is associated with.

Text Channel Component

Contents

This chapter discusses the text sequence grabber channel component and the associated text digitizer components introduced in QuickTime 2.5. Just as video digitizer components obtain digitized video from an analog video source, text digitizer components obtain text from an external source. A text channel component is a sequence grabber channel component. A text digitizer is a new kind of component. The text channel component uses the services of text digitizer components.

For more information about sequence grabber components, see the chapter "Sequence Grabber Components" in *Inside Macintosh: QuickTime Components*. For more information about sequence grabber channel components, see the chapter "Sequence Grabber Channel Components" in *Inside Macintosh: QuickTime Components*.

This chapter is divided into the following major sections:

- "About the Text Channel Component" discusses the characteristics of the QuickTime text channel component and text digitizer components
- "Text Channel Component Reference" describes the functions provided by the QuickTime text channel component

About the Text Channel Component

The QuickTime text channel component allows an application to obtain text from an external source. Once obtained, this text can be previewed or recorded into a QuickTime movie. The source of the text is unknown to the text channel component; a text digitizer component (`'tdig'`) is responsible for acquiring the text from the external source. The text channel component is provided by QuickTime.

Text digitizers are separate components; they are the mechanism for presenting new sources of text data to QuickTime. Several text digitizer components are available, including one that captures closed-captioned data using an Apple TV Tuner card. For more information on creating a component, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*.

To retrieve text for previewing or for recording in a QuickTime movie, the application uses the text channel the same way in which it would use a video channel. The application calls a sequence grabber component, which, in turn, calls the text channel component. The text channel component calls the

appropriate text digitizer component to retrieve the text. For more information on how to use sequence grabber components, see the chapter "Sequence Grabber Components" in *Inside Macintosh: QuickTime Components*.

Once text has been retrieved, the application can request that the sequence grabber component store the text in a text track of a QuickTime movie. For more information on the text media handler, see chapter "Movie Toolbox" in *Inside Macintosh: QuickTime*.

The QuickTime text channel component supports some, but not all, functions defined for sequence grabber channel components and sequence grabber panel components. The supported functions are described in Table 10-1.

In addition, the text channel component provides new functions implemented specifically for text; you use these functions to format captured text to be viewed or added to a text track of a movie. The new functions are described in "Text Channel Component Functions" (page 10-6).

Table 10-1 Functions supported by the text channel component

Usage	Supported

General sequence grabber component functions	`SGSetGWorld`
	`SGNewChannel`
	`SGStartPreview`
	`SGStartRecord`
	`SGIdle`
	`SGStop`
	`SGPause`
	`SGPrepare`
	`SGRelease`
	`SGGetChannelDeviceList`
	`SGUpdate`

Table 10-1 Functions supported by the text channel component (continued)

Usage	Supported
Functions for getting and setting channel characteristics	`SGSetChannelUsage`
	`SGGetChannelUsage`
	`SGSetChannelBounds`
	`SGGetChannelBounds`
	`SGGetChannelInfo`
	`SGSetChannelClip`
	`SGGetChannelClip`
	`SGGetChannelSampleDescription`
	`SGSetChannelDevice`
	`SGSetChannelMatrix`
	`SGGetChannelMatrix`
	`SGGetChannelTimeScale`
Text channel component functions called by sequence grabber components	`SGInitChannel`
	`SGWriteSamples`
	`SGGetDataRate`
Sequence grabber panel component functions	`SGPanelGetDitl`
	`SGPanelInstall`
	`SGPanelEvent`
	`SGPanelRemove`
	`SGPanelGetSettings`
	`SGPanelSetSettings`
	`SGPanelItem`

Text Channel Component Reference

Text Channel Component Functions

This section describes the functions provided by the text channel component for formatting text to be previewed or added to a text track of a movie.

SGSetFontName

The `SGSetFontName` function sets the name of the font to be used to display text.

```
pascal ComponentResult SGSetFontName (
                    SGChannel c,
                    StringPtr pstr);
```

`c` Specifies the channel for this operation.

`pstr` A pointer to a Pascal string containing the name of the font.

DISCUSSION

You call this function to specify a font for the text channel component. If the specified font is available on the system, the text channel uses the font to display text. If the specified font is not available, the text channel uses the default system font. For more information about fonts, see *Inside Macintosh: Text*.

RESULT CODES

Component Manager errors, as documented in *Inside Macintosh: More Macintosh Toolbox*.

SGSetFontSize

The `SGSetFontSize` function sets the font size to be used to display text.

```
pascal ComponentResult SGSetFontSize (
                    SGChannel c,
                    short fontSize);
```

`c` Specifies the channel for this operation.

`fontSize` Specifies the point size of the font.

DISCUSSION

You call this function to specify a text point size for the text channel component. The specified point size must be a positive integer value. For more information about fonts and point size, see *Inside Macintosh: Text*.

RESULT CODES

Component Manager errors, as documented in *Inside Macintosh: More Macintosh Toolbox*.

SGSetTextForeColor

The `SGSetTextForeColor` function sets the color to be used to display text.

```
pascal ComponentResult SGSetTextForeColor (
                    SGChannel c,
                    RGBColor *theColor);
```

`c` Specifies the channel for this operation.

`theColor` Contains a pointer to an `RGBColor` structure that contains the new text color.

DISCUSSION

You call this function to set the text color for a text track.

RESULT CODES

Component Manager errors, as documented in *Inside Macintosh: More Macintosh Toolbox*.

SGSetTextBackColor

The `SGSetTextBackColor` function sets the background color to be used for the text box.

```
pascal ComponentResult SGSetTextBackColor (
                    SGChannel c,
                    RGBColor *theColor);
```

`c`	Specifies the channel for this operation.
`theColor`	Contains a pointer to an `RGBColor` structure that contains the new background color.

DISCUSSION

You call this function to set the background color of a text track. The text channel component uses the specified color as the background of the text box.

RESULT CODES

Component Manager errors, as documented in *Inside Macintosh: More Macintosh Toolbox*.

SGSetJustification

The `SGSetJustification` function sets the alignment to be used to display text.

```
pascal ComponentResult SGSetJustification (
                    SGChannel c,
                    short just);
```

`c`	Specifies the channel for this operation.
`just`	Specifies a constant that represents the text alignment. Possible values are `teFlushDefault`, `teCenter`, `teFlushRight`, and `teFlushLeft`.

DISCUSSION

You call this function, passing a text justification constant, to specify the alignment to be used for text in a text track. The text channel component justifies text relative to the boundaries of its text box. For more information on text alignment and the text justification constants, see the "TextEdit" chapter of *Inside Macintosh: Text*.

RESULT CODES

Component Manager errors, as documented in *Inside Macintosh: More Macintosh Toolbox*.

SGGetTextRetToSpaceValue

The `SGGetTextRetToSpaceValue` function indicates whether the text channel component should replace return characters with spaces.

```
pascal ComponentResult SGGetTextRetToSpaceValue (
                    SGChannel c,
                    short *rettospace);
```

`c`	Specifies the channel for this operation.
`rettospace`	Contains a pointer to a 16-bit integer. On return, this parameter is `true` if the text channel is replacing return characters with spaces, or `false` if the text channel is not replacing return characters with spaces.

DISCUSSION

When you capture text from a closed-caption television source, the text is composed of four lines of text of up to 32 characters each, each line separated

by a return character. Sometimes it is useful to replace the return characters with spaces. You can call this function to determine whether the text channel component is replacing return characters with spaces.

RESULT CODES

Component Manager errors, as documented in *Inside Macintosh: More Macintosh Toolbox*.

SGSetTextRetToSpaceValue

The `SGSetTextRetToSpaceValue` function sets whether the text channel component should replace return characters with spaces.

```
pascal ComponentResult SGSetTextRetToSpaceValue (
                    SGChannel c,
                    short rettospace);
```

`c`	Specifies the channel for this operation.
`rettospace`	Specifies whether return characters should be replaced by spaces. Set this parameter to `true` if the text channel should replace return characters with spaces, or `false` if the text channel should not replace return characters with spaces.

DISCUSSION

When you capture text from a closed-caption television source, the text is composed of four lines of text of up to 32 characters each, each line separated by a return character. You call this function to request that the text channel component replace the return characters with spaces.

RESULT CODES

Component Manager errors, as documented in *Inside Macintosh: More Macintosh Toolbox*.

Movie Data Exchange Components

Contents

This chapter discusses new features in movie data exchange components as documented in Chapter 9 of *Inside Macintosh: QuickTime Components*.

New Features of Movie Data Exchange Components

Exporting Text

The text export and import components provide new features that make it easier to work with the data in a text track in a QuickTime movie. **Text descriptors** are formatting commands that you can embed within a text file. **Time stamps** describe a text sample's starting time and duration.

The text export and import components now make it easier to edit and format text using an external tool, such as a text editor or word processor. When you export text from a text track, you can optionally export text descriptors and time stamps for the text. You can open the text file in a word processor and make changes to the text, style, color, and time stamps. You can then import the edited text to a text track where all the timing , style, color and time stamp information will be present.

When you export text, you control whether text descriptors and time stamps are to be exported by selecting the appropriate options in the Text Export Settings dialog box, shown in Figure 11-1. To display this dialog box programmatically, you call the `MovieExportDoUserDialog` function, described in *Inside Macintosh: QuickTime Components*.

Based on the options you specify in the Text Export Settings dialog box, the text export component is assigned one of three text export option constants: `kMovieExportTextOnly`, `kMovieExportAbsoluteTime`, or `kMovieExportRelativeTime`.

Figure 11-1 Text Export Settings dialog box

If you choose "Show Text Only", the text component is assigned the export option constant `kMovieExportTextOnly`. In this case, the text component exports only text samples, without text descriptors or time stamps. This option is useful when you want to export only the text from a movie and you do not intend to import the text back into a movie.

If you select "Show Text, Descriptors, and Time", the text component is assigned one of two export option constants, depending on the format you specify for time stamps:

- If you specify time stamps to be relative to the start of the movie, the text component is assigned the export option constant `kMovieExportAbsoluteTime`. In this case, the text export component exports text, along with both text descriptors and time stamps. Time stamps are calculated relative to the start of the movie. For example, in exported text with absolute time stamps, the time stamp `[00:00:04.000]` indicates that a text sample begins 4 seconds after the start of the movie.
- If you specify time stamps to be relative to the sample, the text component is assigned the export option constant `kMovieExportRelativeTime`. In this case, the text export component exports text, along with both text descriptors and time stamps. The time stamp for each sample is calculated relative to the end of the previous sample. For example, in exported text with relative time stamps, the time stamp `[00:00:04.000]` indicates that a text sample begins 4

seconds after the beginning of the previous sample. In other words, the previous sample lasts 4 seconds.

For more information about time stamps, see "Time Stamps" (page 11-13).

The text export component provides two functions you can use to access the component's text export option programmatically. To retrieve the current value of the text export option, you call the `TextExportGetSettings` function (page 11-22). To set the value of the text export option, you call the `TextExportSetSettings` function (page 11-23).

The Text Export Settings dialog box also allows you to specify the time scale the text component uses to specify the fractional part of a time stamp. The value should be between 1 and 10000, inclusive. The text export component provides two functions you can use to access the component's time scale programmatically. To retrieve the time scale, you call the `TextExportGetTimeFraction` function (page 11-20). To set the time scale, you call the `TextExportSetTimeFraction` function (page 11-21).

Text Descriptors

A text descriptor is a formatting command that describes the text that follows it. Exporting text with text descriptors allows you to edit text from a text track, including its formatting, in an external program, such as a text editor or word processor. When you import the edited text, the formatting you specified with the text descriptors is preserved. This provides an easy way to localize movies for different languages, correct spelling, change styles, or modify text behavior.

A text descriptor has the following format:

{*descriptor*}

For example, the text descriptor `{bold}` sets the text in the current text sample and all subsequent text samples. Some text descriptors, such as `{bold}`, have no parameters. Other text descriptors have one or more parameters. For text descriptors with parameters, the descriptor is followed by a colon and its parameters, separated by commas. You can specify text descriptors using either uppercase or lowercase characters, with or without spaces separating the parameters:

{*descriptor*: *parameter1*, ..., *parameterN*}

For example, the text descriptor `{font:New York}` sets the text in the current text sample and all subsequent text samples to the New York font. The New York font is applied to all text until a second `{font:}` text descriptor is issued.

A text stream that contains text descriptors and time stamps should always begin with the text descriptor `{QTtext}`, followed by any number of text descriptors in any order. If the text import component detects a typographical error inside a descriptor while importing a text file, it may generate partial results or an error message stating that the text file cannot be converted.

When text with text descriptors is imported into a track, the information represented by the descriptors is stored in a text display data structure (type `TextDisplayData`). Text descriptors whose possible values are `on` and `off` are used to set flags in the `displayFlags` field of the text display data structure. Each sample in the text track has a corresponding text display data structure that contains the text attributes of the sample. For more information, see "Text Display Data Structure" (page 11-17).

Listing 11-1 shows a simple example of text that has been exported with text descriptors and time stamps. The text sample displays the text "I ♥ Apple" (the "© "character is a "♥" in the Symbol font). The background color is black and the text color is white, except for the "♥" character, which is drawn in red.

Listing 11-1 Formatting text using text descriptors

```
{QTtext} {font:New York} {plain} {size:36} {textColor: 65535, 65535,
65535} {backColor: 0, 0, 0} {justify:center} {timeScale:600} {width:320}
{height:0} {timeStamps:absolute}
[00:00:00.000]
I {font:Symbol}{size:46}{textColor:65535, 0, 0}© {textColor: 65535,
65535, 65535}{font:New York} {plain} {size:36} Apple
[00:00:05.300]
```

The `{karaoke:}` text descriptor allows you to highlight groups of characters in a text sample at specified times. For example, you might want to highlight the words in a song while playing the song's sound track. The parameters for the `{karaoke:}` text descriptor are specified as a set of tuples, separated by semicolons. The set of tuples is preceded by the total number of tuples:

`{karaoke:` *count*`;` *a1, a2, a3; b1, b2, b4; ... n1, n2, n3*`}`*text sample*

Each tuple is composed of a time value (that is greater then the one specified and less then the next time value), a starting offset into the text, and an ending offset into the text; at the specified time value, the text from the starting offset to the ending offset is highlighted.

The following example shows a `{karaoke:}` text descriptor followed by some text. The time scale of the movie was set to 600. The `{karaoke:}` text descriptor has 14 tuples. The second tuple indicates that, between 125/600 seconds and 250/600 seconds, the text from offset 0 to offset 4 ("`Thun`") should be highlighted. The third tuple indicates that, after 250/600 seconds and before 350/600 seconds, the text from offset 4 to offset 7 ("`der`") should be highlighted.

```
{karaoke: 14; 0, 0, 0; 125, 0, 4; 250, 4, 7; 375, 7, 11; 500, 12, 15;
750, 16, 21; 1000, 21, 25; 1125, 25, 28; 1250, 28, 30; 1375, 31, 33;
1500, 33, 35; 1750, 36, 42; 2000, 42, 47; 2250, 48, 50;}Thunderbolt and
lightning very very fright'ning me
```

The text export and import components support the following text descriptors:

General

`{QTtext}` Required at the beginning of any text file that contains descriptors or time stamps. If the text import component does not detect this descriptor at the beginning of the file, it assumes the file is a standard text file and will not look for descriptors or time stamps.

`{language:}` Specifies the language of the text track. This text descriptor takes one parameter, the ordinal value of the language. For example, `{language:11}` specifies that the language of the track is Japanese. For more information on language ordinal values, see the chapter "Movie Resource Formats" in *Inside Macintosh: QuickTime*.

Text styles

`{font:}` Specifies the name of the font to be used. For example, the text descriptor `{font:Apple Chancery}` changes the font to Apple Chancery.

`{size:}` Specifies the point size of the text. For example, the text descriptor `{size:18}` sets the text point size to 18 points.

{plain}	Resets the text style. This text descriptor resets the following text descriptors: {bold}, {italic}, {underline}, {shadow}, {outline}, {extend}, and {condense}.
{bold}	Specifies bold text.
{italic}	Specifies italic text.
{underline}	Specifies underlined text.
{outline}	Specifies outlined text.
{shadow}	Specifies shadow text.
{condense}	Specifies condensed text.
{extend}	Specifies extended text.

Text dimensions

{height:}	Specifies the height of the text track in pixels. For example, the text descriptor {height:50} sets the text track height to 50 pixels. The text descriptor {height:0} sets the height to the best height for the text.
{width:}	Specifies the width of the text track in pixels. For example, the text descriptor {width:50} sets the text track width to 50 pixels. The text descriptor {width:0} sets the width to an appropriate default or, if importing a movie, to the width of the movie.
{textBox:}	Specifies the coordinates of the text box. This text descriptor takes four parameters: top, left, bottom, and right. For example, if you specify {textBox:0, 0, 80, 320}, the text box originates at (0,0) and is 320 pixels wide and 80 pixels high.
{doNotAutoScale:}	Specifies whether to automatically scale the text if the track bounds increase. This text descriptor takes one parameter. Possible values are on and off; the default value is off. For example, the text descriptor {doNotAutoScale:off} enables auto-scaling. This corresponds to the dfDontAutoScale display flag.
{clipToTextBox:}	Specifies whether to clip to the text box. This is useful if the text overlays the video. This text descriptor takes one parameter. Possible values are on and off; the default value is off. This corresponds to the dfClipToTextBox display flag.

`{shrinkTextBox:}` Specifies whether to recalculate the size of the text box specified by the `{textBox:}` text descriptor to fit the dimensions of the text. If so, the new rectangle is stored with the text data. This text descriptor takes one parameter. Possible values are `on` and `off`. The value of this text descriptor is used to set a flag in the `displayFlags` field of the text display data structure. This corresponds to the `dfShrinkTextBoxToFit` display flag.

Drawing text

`{doNotDisplay:}` Specifies whether to display the sample. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. For example, the text descriptor `{doNotDisplay:on}` causes the sample to not be displayed. This corresponds to the `dfDontDisplay` display flag.

`{justify:}` Specifies the alignment of the text in the text box. This text descriptor takes one parameter. Possible values are `left`, `right`, `center`, and `default`. For example, the text descriptor `{justify:left}` aligns text on the left. For more information on text alignment, see the chapter "TextEdit" in *Inside Macintosh: Text*.

`{anti-alias:}` Specifies whether text should be displayed using anti-aliasing. Anti-aliasing smooths the edges of the text by blending the edge colors of the text and background. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. This corresponds to the `dfAntiAlias` display flag.

`{textColor:}` Specifies the color of the text. This text descriptor takes three parameters: red, green, and blue color values. Each parameter should be between 0 and 65535. For example, the text descriptor `{textColor:45000,0,0}` sets the text color to a shade of red.

`{backColor:}` Specifies the background color of the region specified by the `{textBox:}` text descriptor or the region specified by the `{height:}` and `{width:}` descriptors. This text descriptor takes three parameters: red, green, and blue color values. Each parameter should be between 0 and 65535. For example, the text descriptor

`{backColor:0,45000,0}` sets the background color to a shade of green.

`{hiliteColor:}` This display flag specifies the color to be used to highlight text. This text descriptor takes three parameters: red, green, and blue color values. Each parameter should be between 0 and 65535. For example, the text descriptor `{hiliteColor:45000,0,0}` sets the highlight color to a shade of red.

`{inverseHilite:}` This display flag specifies whether to highlight text using reverse video instead of the highlight color. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. This corresponds to the `dfInverseHilite` display flag.

`{keyedText:}` This display flag specifies whether text should be rendered over the background without drawing the background color. This technique is also known as masked text. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. This corresponds to the `dfKeyedText` display flag.

`{hilite:}` Specifies characters to be highlighted in a text sample. This text descriptor takes two parameters, the first and last characters to highlight in the sample. For example, the text descriptor `{hilite: 11, 14}` highlights the word "text" in the text sample "This is a text track".

`{textColorHilite:}` This display flag specifies whether text should be highlighted by changing the color of the text. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`.

`{karaoke:}` Highlights groups of characters in the subsequent text sample at specified times. For example, you use this text descriptor to highlight the words in a song while playing the song's sound track. The parameters for this text descriptor are specified as a set of tuples of the form (*time value*, *starting offset*, *ending offset*), separated by semicolons. The set of tuples is preceded by the total number of tuples. For each tuple, at the specified time value, the text from the starting offset to the ending offset is highlighted.

`{continuousKaraoke:}` This display flag specifies whether karaoke should ignore

the starting offset and highlight all text from the beginning of the sample to the ending offset. Possible values are `on` and `off`; the default value is `off`. If continuous karaoke is not enabled, karaoke highlights one offset range at a time. In order for this text descriptor to take effect, the karaoke text descriptor must be in effect. This corresponds to the `dfContinuousKaraoke` display flag.

`{dropShadow:}` This display flag specifies whether the text sample supports drop shadows. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. This corresponds to the `dfDropShadow` display flag.

`{dropShadowOffset:}` Specifies an offset for the drop shadow. This text descriptor takes two parameters, an offset to the right and an offset down. For example, the text descriptor `{dropShadowOffset: 3, 4}` offsets the drop shadow 3 pixels to the right and 4 pixels down. The default is `{dropShadowOffset: 6, 6}`. In order for this text descriptor to take effect, drop shadowing must be enabled.

`{dropShadowTransparency:}`
Specifies the intensity of the drop shadow. This text descriptor takes one parameter, a value between 0 and 255. The default value is 127. In order for this text descriptor to take effect, drop shadowing must be enabled.

Time stamps

`{timeStamps:}` Specifies whether time stamps are absolute or relative. This text descriptor takes one parameter. Possible values are `absolute` and `relative`. For example, `{timestamps:absolute}` specifies absolute time stamps. If time stamps are absolute, for each sample, the time stamp is specified relative to the start of the track. If time stamps are relative, for each sample, the time stamp is specified relative to the previous time stamp.

`{timeScale:}` Specifies the time scale for the text track. This time scale is used to calculate the mantissa for a time stamp. For example, if the text descriptor `{timeScale:600}` is specified, the time stamp [00:00:07.300] would be interpreted as 7.5 seconds. The maximum value for this parameter is 10000.

Scrolling

`{scrollIn:}` This display flag specifies whether text should be scrolled in until the last of the text is in view. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. If both `{scrollIn:}` and `{scrollOut:}` are set, the text is scrolled in and then scrolled out. If the `{scrollDelay:}` text descriptor is set, text is scrolled using the specified delay. This corresponds to the `dfScrollIn` display flag.

`{scrollOut:}` This display flag specifies whether text should be scrolled out until the last of the text is in view. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. If both `{scrollIn:}` and `{scrollOut:}` are set, the text is scrolled in and then scrolled out. If the `{scrollDelay:}` text descriptor is set, text is scrolled using the specified delay. This corresponds to the `dfScrollOut` display flag.

`{horizontalScroll:}` This display flag specifies whether to scroll a single line of text horizontally. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. If you do not specify this descriptor, the scrolling is vertical. The scrolling direction is determined by the `{reverseScroll:}` text descriptor. This corresponds to the `dfHorizScroll` display flag.

`{reverseScroll:}` This display flag specifies whether to reverse the direction of scrolling. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. For vertical scrolling, the default direction is up. For horizontal scrolling, the default direction is left. For example, if you specify `{reverseScroll:on}` and `{horizontalScroll:off}`, the text is scrolled vertically down. This corresponds to the `dfReverseScroll` display flag.

`{continuousScroll:}` This display flag specifies whether new samples should cause previous samples to scroll out. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. In order for this text descriptor to take effect, either `{scrollIn:}` or `{scrollOut:}` must be enabled. This corresponds to the `dfContinuousScroll` display flag.

`{flowHorizontal:}`	This display flag specifies whether text flows within the text box when it is scrolled horizontally. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. For example, if you specify `{flowHorizontal:off}`, the text flows as if the text box had no right edge. In order for this text descriptor to take effect, horizontal scrolling must be enabled. This corresponds to the `dfFlowHoriz` display flag.
`{scrollDelay:}`	This display flag specifies a scroll delay for the sample. This text descriptor takes one parameter, the number of units of the delay in the text track's time scale. For example, if the time scale is 600, the text descriptor `{scrollDelay: 600}` causes subsequent text to be delayed one second. In order for this text descriptor to take effect, scrolling must be enabled.

Time Stamps

When you export text and text descriptors from a text track, the text component also exports a time stamp for each sample. The time stamp indicates the starting and ending time of the sample, either relative to the start of the movie (`kMovieExportAbsoluteTime`) or to the end of the previous sample (`kMovieExportRelativeTime`). On import, the time stamps maintain the timing positions of the text samples relative to other media in the movie.

The format of a time stamp is

```
[HH:MM:SS.xxx]
```

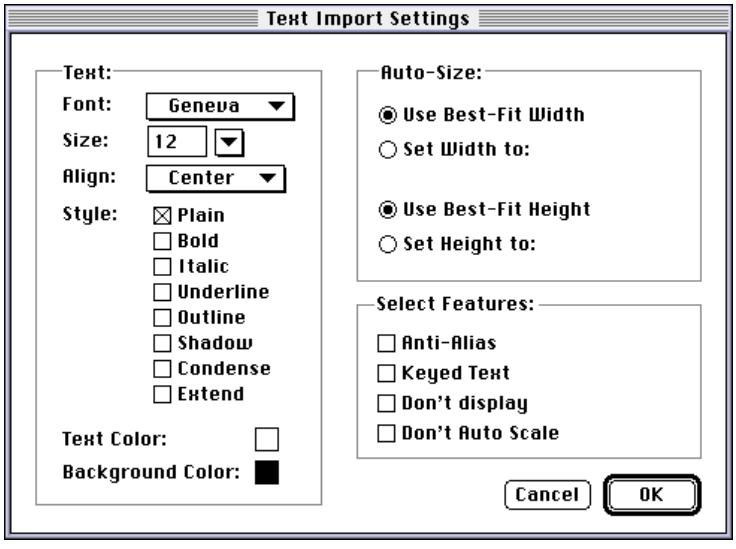
where `HH` represents the number of hours, `MM` represents the number of minutes, `SS` represents the number of seconds, and `xxx` represents the mantissa (the fractional part of a second). The mantissa is expressed in the time scale of the text track. For example, if the time scale of the text track is 600, the time stamp `[00:00:07.300]` is interpreted as 7.5 seconds. If the time scale of the text track is 10, the time stamp `[00:00:07.5]` is also interpreted as 7.5 seconds. The maximum time scale for a text track is 10000.

When a text export component exports a text sample, it first exports the time stamp, followed by a Return character. Then, it exports the sample's text and text descriptors. If a text sample does not contain any text, the text component exports the time stamp and Return character, but no text.

Importing Text

When you import text, you can override the text descriptors in the text file by specifying options in the Text Import Settings dialog box, shown in "Text Import Settings dialog box". On import, the settings specified in the dialog box are applied to all imported samples. To display this dialog box programmatically, you call the `MovieImportDoUserDialog` function, described in *Inside Macintosh: QuickTime Components*.

Figure 11-2 Text Import Settings dialog box



Importing In Place

Some movie data import components can create a movie from a file without having to write to a separate disk file. Examples include MPEG and AIFF import components—data in files of these types can be played directly by the appropriate media handler components, without any data conversion. In such cases it is inappropriate for the user to have to specify a destination file, given that there is no need for such a file.

If your import component can operate in this manner, set the `canMovieImportInPlace` flag to 1 in your component flags when you register

your component. The standard file dialog uses this flag to determine how to import files. The `OpenMovieFile` and `NewMovieFromFile` functions use this flag to open some kinds of files as movies.

Audio CD Import Component

QuickTime 1.6.1 introduced the audio CD import component. This movie import component allowed users to open audio CD tracks from the QuickTime Standard File Preview dialog, then convert and save the audio as a movie.

In QuickTime 2.1, the audio CD import component was revised to create AIFF files, rather than movie files. These files also contain movie resources, so you can open them as movies.

When you open an audio track on an Apple CD-ROM drive (or equivalent), the Open button changes to a Convert button. When you click Convert, the Audio CD Import Options dialog appears. Use this dialog box to configure the sound settings. You can specify the sample rate, sample size, and channel settings. You can also select the portion of the track that you want to convert.

Movie Data Exchange Components Reference

This section describes new constants, data types, and functions of movie data exchange components.

Constants

Flags for Movie Import and Export Components

QuickTime 1.6.1 added four new `componentFlags` values. The `canMovieImportInPlace` and `movieImportSubTypeIsFileExtension` values were added in QuickTime 2.0:

```
enum {
    canMovieExportAuxDataHandle     = 1 << 7,
    canMovieImportValidateHandles   = 1 << 8,
```

```
    canMovieImportValidateFile      = 1 << 9,
    dontRegisterWithEasyOpen        = 1 << 10,
    canMovieImportInPlace           = 1 << 11,
    movieImportSubTypeIsFileExtension = 1 << 12
};
```

Constant descriptions

`canMovieExportAuxDataHandle`
: Set this bit to export a movie to an auxillary data handle. A movie export component that supports the `MovieExportGetAuxillaryData` function should also set this flag in its `componentFlags`.

`canMovieImportValidateHandles`
: Set this bit if your movie import component can and wants to validate handles.

`canMovieImportValidateFile`
: Set this bit if your movie import component can and wants to validate files.

`dontRegisterWithEasyOpen`
: Set this bit if Macintosh Easy Open is installed and you do not want to register your component and you want to handle interactions with Macintosh Easy Open yourself.

`canMovieImportInPlace`
: Set this bit if your movie import component can create a movie from a file without having to write to a separate disk file. Examples include MPEG and AIFF import components.

`movieImportSubTypeIsFileExtension`
: Set this bit if your component's subtype is a file extension instead of a Macintosh file type. For example, if your import component opens files with an extension of `.doc`, you would set this flag and set your component subtype to `'DOC '`.

Text Export Options

The following constants represent options for exporting text using a text export component. You use these constants to specify the format of the exported text. From the QuickTime user interface, you specify a text export option in the Text

Export Settings dialog box. You can also specify the text export option programmatically by calling the `TextExportSetSettings` function (page 11-23).

```
enum {
    kMovieExportTextOnly    = 0,
    kMovieExportAbsoluteTime= 1,
    kMovieExportRelativeTime= 2
};
```

Constant descriptions

`kMovieExportTextOnly`
Export text only, without text descriptors or time stamps.

`kMovieExportAbsoluteTime`
Export text with text descriptors and time stamps. For each sample, calculate the time stamp relative to the start of the movie.

`kMovieExportRelativeTime`
Export text with text descriptors and time stamps. For each sample, calculate the time stamp relative to the previous time stamp.

Data Types

Text Display Data Structure

The `TextDisplayData` structure contains formatting information for a text sample. When the text export component exports a text sample, it uses the information in this structure to generate the appropriate text descriptors for the sample. Likewise, when the text import component imports a text sample, it sets the appropriate fields in the text display data structure based on the sample's text descriptors.

```
struct TextDisplayData {
    long                    displayFlags;
    long                    textJustification;
    RGBColor                bgColor;
    Rect                    textBox;
    short                   beginHilite;
```

```
    short                   endHilite;
    RGBColor                hiliteColor;
    Boolean                 doHiliteColor;
    SInt8                   filler;
    TimeValue               scrollDelayDur;
    Point                   dropShadowOffset;
    short                   dropShadowTransparency;
};
typedef struct TextDisplayData TextDisplayData;
```

Field descriptions

`displayFlags` Contains flags that represent the values of the following text descriptors: `doNotDisplay`, `doNotAutoScale`, `clipToTextBox`, `useMovieBackColor`, `shrinkTextBox`, `scrollIn`, `scrollOut`, `horizontalScroll`, `reverseScroll`, `continuousScroll`, `flowHorizontal`, `dropShadow`, `anti-alias`, `keyedText`, `inverseHilite`, `continuousKaraoke`, and `textColorHilite`. For more information on the text sample display flags, see Chapter 1, "Movie Toolbox," in this book and the description of the `AddTextSample` function in *Inside Macintosh: QuickTime*.

`textJustification` Specifies the alignment of the text in the text box. Possible values are `teFlushDefault`, `teCenter`, `teFlushRight`, and `teFlushLeft`. For more information on text alignment and the text justification constants, see the "TextEdit" chapter of *Inside Macintosh: Text*.

`bgColor` Specifies the background color of the rectangle specified by the `textBox` field. The background color is specified as an RGB color value.

`textBox` Specifies the rectangle of the text box.

`beginHilite` Specifies the one-based index of the first character in the sample to highlight.

`endHilite` Specifies the one-based index of the last character in the sample to highlight.

`doHiliteColor` Specifies whether to use the color specified by the `hiliteColor` field for highlighting. If the value of this field is `true`, the highlight color is used for highlighting. If the value of this field is `false`, reverse video is used for highlighting.

`filler`	Reserved.
`scrollDelayDur`	Specifies a scroll delay. The scroll delay is specified as the number of units of delay in the text track's time scale. For example, if the time scale is 600, a scroll delay of 600 causes the sample text to be delayed one second. In order for this field to take effect, scrolling must be enabled.
`dropShadowOffset`	Specifies an offset for the drop shadow. For example, if the point specified is (3,4), the drop shadow is offset 3 pixels to the right and 4 pixels down. In order for this field to take effect, drop shadowing must be enabled.
`dropShadowTransparency`	Specifies the intensity of the drop shadow as a value between 0 and 255. In order for this field to take effect, drop shadowing must be enabled.

Movie Data Exchange Components Functions

Exporting Text

This section describes new functions provided by text export components.

TextExportGetDisplayData

The `TextExportGetDisplayData` function retrieves text display information for the current sample in the specified text export component.

```
pascal ComponentResult TextExportGetDisplayData (
                    TextExportComponent ci,
                    TextDisplayData *textDisplay);
```

`ci` Specifies the text export component for this operation. Applications obtain this reference from the Component Manager's `OpenComponent` function.

`textDisplay` Contains a pointer to a text display data structure. On return, this structure contains the display settings of the current text sample. For more information, see "Text Display Data Structure" (page 11-17).

DISCUSSION

You call this function to retrieve the text display data structure for a text sample. The text display data structure contains the formatting information for the text sample. When the text export component exports a text sample, it uses the information in this structure to generate the appropriate text descriptors for the sample. Likewise, when the text import component imports a text sample, it sets the appropriate fields in the text display data structure based on the sample's text descriptors.

RESULT CODES

Component Manager errors, as documented in *Inside Macintosh: More Macintosh Toolbox*.

TextExportGetTimeFraction

The `TextExportGetTimeFraction` function retrieves the time scale the specified text export component uses to calculate time stamps.

```
pascal ComponentResult TextExportGetTimeFraction (
                    TextExportComponent ci,
                    long *movieTimeFraction);
```

`ci` Specifies the text export component for this operation.

`movieTimeFraction`
Contains a pointer to a 32-bit integer. On return, this integer contains the time scale used in the fractional part of time stamps.

DISCUSSION

You call this function to retrieve the time scale used by the text export component to calculate the fractional part of time stamps. You set a text component's time scale by specifying it in the Text Export Settings dialog or by calling the `TextExportSetTimeFraction` function.

RESULT CODES

Component Manager errors, as documented in *Inside Macintosh: More Macintosh Toolbox*.

TextExportSetTimeFraction

The `TextExportSetTimeFraction` function set the time scale the specified text export component uses to calculate time stamps.

```
pascal ComponentResult TextExportSetTimeFraction (
                    TextExportComponent ci,
                    long movieTimeFraction);
```

`ci` Specifies the text export component for this operation.

`movieTimeFraction`
Specifies the time scale used in the fractional part of time stamps. The value should be between 1 and 10000, inclusive.

DISCUSSION

You call this function to set the time scale used by the text export component to calculate the fractional part of time stamps. You can also set a text component's time scale by specifying it in the Text Export Settings dialog. You can retrieve a text component's time scale by calling the `TextExportGetTimeFraction` function.

RESULT CODES

Component Manager errors, as documented in *Inside Macintosh: More Macintosh Toolbox*.

TextExportGetSettings

The `TextExportGetSettings` function retrieves the value of the text export option for the specified text export component.

```
pascal ComponentResult TextExportGetSettings (
                    TextExportComponent ci,
                    long *setting);
```

`ci` Specifies the text export component for this operation.

`setting` Contains a pointer to a 32-bit integer. On return, this integer contains a constant that represents the current value of the text export option. Possible values are `kMovieExportTextOnly`, `kMovieExportAbsoluteTime`, and `kMovieExportRelativeTime`. For more information, see "Text Export Options" (page 11-16).

DISCUSSION

You call this function when exporting text to retrieve the current value of the text export option for the specified text export component. If the retrieved text export option is `kMovieExportTextOnly`, the text export component exports text without time descriptors or time stamps. If the retrieved text export option is either `kMovieExportAbsoluteTime` or `kMovieExportRelativeTime`, the text export component exports text along with its text descriptors and time stamps.

RESULT CODES

Component Manager errors, as documented in *Inside Macintosh: More Macintosh Toolbox*.

TextExportSetSettings

The `TextExportSetSettings` function sets the value of the text export option for the specified text export component.

```
pascal ComponentResult TextExportSetSettings (
                    TextExportComponent ci,
                    long setting);
```

`ci` Specifies the text export component for this operation.

`setting` Specifies a constant that represents the current value of the text export option. Possible values are `kMovieExportTextOnly`, `kMovieExportAbsoluteTime`, and `kMovieExportRelativeTime`. For more information, see "Text Export Options" (page 11-16).

DISCUSSION

You call this function when exporting text to set the value of the text export option for the specified text export component. To export text only, without time descriptors or time stamps, you should set the `setting` parameter to `kMovieExportTextOnly`. To export text with text descriptors and absolute time stamps, you should set the `setting` parameter to `kMovieExportAbsoluteTime`. To export text with text descriptors and relative time stamps, you should set the `setting` parameter to `kMovieExportRelativeTime`.

RESULT CODES

Component Manager errors, as documented in *Inside Macintosh: More Macintosh Toolbox*.

Importing Movie Data

This section describes new movie import component functions.

MovieImportGetFileType

The `MovieImportGetFileType` allows your movie data import component to tell the Movie Toolbox the appropriate file type for the most-recently imported movie file.

```
pascal ComponentResult MovieImportGetFileType (MovieImportComponent ci,
                    OSType *fileType);
```

`ci` Identifies the Movie Toolbox's connection to your movie data import component.

`fileType` Contains a pointer to an `OSType` field. Your component should place the file type value that best identifies the movie data just imported. For example, Apple's Audio CD movie data import component sets this field to `'AIFF'` whenever it creates an AIFF file instead of a movie file.

DISCUSSION

You should implement this function only if your movie data import component creates files other than QuickTime movies. By default, the Movie Toolbox makes new files movies, unless you override that default by providing this function.

RESULT CODES

`badComponentSelector`	0x80008002	Function not supported

MovieImportGetAuxiliaryDataType

The `MovieImportGetAuxiliaryDataType` function returns the type of the auxillary data that it can accept. For example, calling the text import component with

`MovieImportGetAuxiliaryDataType` indicates that the text import component will use `'styl'` information in addition to `'text'` data.

```
pascal ComponentResult MovieImportGetAuxiliaryDataType(
                    MovieImportComponent ci,
                    OSType *auxType)
```

`ci`	Specifies the movie import component for the request. Applications obtain this reference from the Component Manager's `OpenComponent` function.
`auxType`	A pointer to the type of auxillary data it can import.

RESULT CODES

`badComponentInstance`	0x80008001	Invalid movie import component instance
`badComponentSelector`	0x80008002	Function not supported

MovieImportValidate

The `MovieImportValidate` function allows your movie data import component to validate the data to be passed to your component.

```
pascal ComponentResult MovieImportValidate(
                    MovieImportComponent ci,
                    const FSSpec *theFile,
                    Handle theData,
                    Boolean *valid)
```

`ci`	Specifies the movie import component for the request. Applications obtain this reference from the Component Manager's `OpenComponent` function.
`theFile`	Specifies the file to validate.
`theData`	Specifies the data to validate.
`valid`	Contains a pointer to a Boolean value. If the data and/or file is valid, this value returns `true`. Otherwise, it returns `false`.

DISCUSSION

Movie import components can implement this function to allow applications to determine if a given file or handle to data is acceptable for a particular import component. As this function may be called on many files, the validation process should be as fast as possible.

RESULT CODES

`badComponentInstance`	0x80008001	Invalid movie import component instance
`badComponentSelector`	0x80008002	Function not supported

Exporting Movie Data

Since QuickTime 1.6.1, the sound movie export component has been updated to take advantage of Sound Manager 3.0. Previously, only the first sound track in the movie was exported. Now sound tracks are mixed together before they are exported. If you want to use sound mixing, you can use the `PutMovieIntoTypedHandle` function to take advantage of the export component. Furthermore, you can now specify the format of the exported sound, so you can convert 16-bit sound to 8-bit sound or reduce stereo to mono. See *Inside Macintosh: QuickTime* for a description of the `PutMovieIntoTypedHandle` function.

Configuring Movie Data Export Components

This section describes new and modified movie export component functions.

MovieExportGetAuxillaryData

QuickTime 1.6.1 added a new result code to this function. A movie export component returns the following result code when `MovieExportGetAuxillaryData` is called requesting a type of auxillary data that the component cannot generate.

RESULT CODES

auxillaryExportDataUnavailable	-2058	Cannot generate the requested type of auxillary data.

MovieExportSetSampleDescription

The MovieExportSetSampleDescription function allows your application to request the format of the exported data. This function is supported by the sound movie export component, for example.

```
pascal ComponentResult MovieExportSetSampleDescription(
                    MovieExportComponent ci,
                    SampleDescriptionHandle desc,
                    OSType mediaType)
```

ci	Specifies the movie component for the request. Applications obtain this reference from the Component Manager's OpenComponent function.
desc	Contains a handle to a valid QuickTime sample description.
mediaType	Indicates the type of media the sample description is for. For example, if the sample description was a sound description, this parameter would be set to SoundMediaType.

DISCUSSION

A movie export component may use all, some, or none of the settings from the sample description.

RESULT CODES

badComponentInstance	0x80008001	Invalid movie import component instance
badComponentSelector	0x80008002	Function not supported

Derived Media Handler Components

Contents

This chapter discusses the changes to derived media handler components as documented in Chapter 10 of *Inside Macintosh: QuickTime Components*.

Derived Media Handler Components Reference

Constants

Media Video Parameters

The `whichparam` parameter to the `MediaSetVideoParam` and `MediaGetVideoParam` functions specifies which video parameter you want to adjust. QuickTime defines these constants that you can use to configure the `whichparam` parameter.

```
enum {
    kMediaVideoParamBrightness  = 1,
    kMediaVideoParamContrast    = 2,
    kMediaVideoParamHue         = 3,
    kMediaVideoParamSharpness   = 4,
    kMediaVideoParamSaturation  = 5,
    kMediaVideoParamBlackLevel  = 6,
    kMediaVideoParamWhiteLevel  = 7
};
```

Constant descriptions

`kMediaVideoParamBrightness`
: The brightness value controls the overall brightness of the digitized video image. Brightness values range from 0 to 65,535, where 0 is the darkest possible setting and 65,535 is the lightest possible setting.

`kMediaVideoParamContrast`
: The contrast value ranges from 0 to 65,535, where 0 represents no change to the basic image and larger values increase the contrast of the video image (that is, increase the slope of the transform).

`kMediaVideoParamHue`

Hue is similar to the tint control on a television, and it is specified in degrees with complementary colors set 180 degrees apart (red is 0°, green is +120°, and blue is –120°). Supports hue values that range from 0 (–180° shift in hue) to 65,535 (+179° shift in hue), where 32,767 represents a 0° shift in hue.

`kMediaVideoParamSharpness`

The sharpness value ranges from 0 to 65,535, where 0 represents no sharpness filtering and 65,535 represents full sharpness filtering. Higher values result in a visual impression of increased picture sharpness

`kMediaVideoParamSaturation`

The saturation value controls color intensity. For example, at high saturation levels, red appears to be red; at low saturation, red appears pink. Valid saturation values range from 0 to 65,535, where 0 is the minimum saturation value and 65,535 specifies maximum saturation.

`kMediaVideoParamBlackLevel`

Black level refers to the degree of blackness in an image.The highest setting produces an all-black image; on the other hand, the lowest setting yields little, if any, black even with black objects in the scene. Black level values range from 0 to 65,535, where 0 represents the maximum black value and 65,535 represents the minimum black value.

`kMediaVideoParamWhiteLevel`

White level refers to the degree of whiteness in an image. White level values range from 0 to 65,535, where 0 represents the minimum white value and 65,535 represents the maximum white value

Data Types

The `GetMovieCompleteParams` data type, which defines the layout of the complete movie parameter structure used by the `MediaInitialize` function, has a new parameter, `inputMap`, which is a reference to the media's input map. The media input map should not be modified or disposed. Because of this change,

the `version` field of the `GetMovieCompleteParams` data type has been changed from 0 to 1.

Derived Media Handler Component Functions

Managing Your Media Handler Component

This section describes functions that apply to all derived media handler components.

MediaIdle

There is a minor change to the `MediaIdle` function that is related to the new media handler support for partial screen redrawing (for more information on this feature see the discussion of the `MediaGetDrawingRgn` function elsewhere in this chapter).

From time to time, your derived media handler component may determine that only a portion of the available drawing area needs to be redrawn. You can signal that condition to the base media handler component by setting the `mPartialDraw` flag to 1 in the flags your component returns to the Movie Toolbox from your `MediaIdle` function. You return these flags using the `flagsOut` parameter.

Whenever you set this flag to 1, the Movie Toolbox calls your component's `MediaGetDrawingRgn` function in order to determine the portion of the image that needs to be redrawn.

As an example, consider a full-screen animation. Only rarely is the entire image in motion. Typically, only a small portion of the screen image moves. By using partial redrawing, you can significantly improve the playback performance of such a movie.

General Data Management

This section contains several new functions to support modifier tracks, active segments, and video settings. These functions apply to all derived media handler components.

MediaGSetActiveSegment

The `MediaGSetActiveSegment` function informs your derived media handlers of the current active segment.

```
pascal ComponentResult MediaGSetActiveSegment(
                    MediaHandler mh,
                    TimeValue activeStart,
                    TimeValue activeDuration)
```

`mh` Identifies the Movie Toolbox's connection to your derived media handler.

`activeStart` Contains the starting time of the active segment to play. This time value is expressed in your movie's time scale.

`activeDuration`
Contains a time value that specifies the duration of the active segment. This value is expressed in the movie's time scale.

DISCUSSION

Using the `SetMovieActiveSegment` function, an application can limit the time segment of the movie that will be used for play back. Derived media handlers are given the values for the active segment by the `MediaGSetActiveSegment` function called by the Movie Toolbox. Active segment information is usually only needed by media handler's that perform their own scheduling.

MediaInvalidateRegion

The `MediaInvalidateRegion` function updates the invalidated display region the next time `MediaIdle` is called.

```
pascal ComponentResult MediaInvalidateRegion(
                    MediaHandler mh,
                    RgnHandle invalRgn)
```

`mh` Identifies the Movie Toolbox's connection to your derived media handler.

`invalRgn` Contains a handle to a region that has been invalidated. Your media handler should not dispose or modify this region. The `invalRgn` parameter will never be `nil`.

DISCUSSION

The `MediaInvalidateRegion` function is called by the Movie Toolbox when `UpdateMovie` or `InvalidateMovieRegion` is called with a region that intersects your media's track.

Derived media handlers will need to implement `MediaInvalidateRegion` only if they can perform efficient updates on a portion of their display area.

If a media handler implements the `MediaInvalidateRegion` function, it is responsible for ensuring that the appropriate areas of the screen are updated on the next call to `MediaIdle`. If a media handler does not implement this function, the base media handler will set the `mMustDrawflag` the next time `MediaIdle` is called.

MediaGetNextStepTime

The `MediaGetNextStepTime` function searches for the next forward or backward **step time** from the given media time. The step time is the time of the next

frame. This function allows a derived media handler to return the next step time from the specified media time.

```
pascal ComponentResult MediaGetNextStepTime(
                    MediaHandler mh,
                    short flags,
                    TimeValue mediaTimeIn,
                    TimeValue *mediaTimeOut,
                    Fixed rate)
```

`mh` Identifies the Movie Toolbox's connection to your derived media handler.

`flags` The following `interestingTimeFlags` flags are defined:

`nextTimeStep` Searches for the next frame in the media. Set this flag to 1 to search for the next frame.

`nextTimeEdgeOk` Instructs the Movie Toolbox that you are willing to receive information about elements that begin or end at the time specified by the `mediaTimeIn` parameter. Set this flag to 1 to accept this information. This flag is especially useful at the beginning or end of a media. The function returns valid information about the beginning and end of the media.

`mediaTimeIn` Specifies a time value that establishes the starting point for the search. This time value is in the media's time scale.

`mediaTimeOut` The step time calculated by the media handler. The media handler should return the first time value it finds that meets the search criteria specified in the `flags` parameter. This time value is in the media's time scale.

`rate` Contains the search direction. Negative values search backward from the starting point specified in the `mediaTimeIn` parameter. Other values cause a forward search.

DISCUSSION

The mechanism in QuickTime used for stepping backwards and forwards a frame at a time are the interesting time calls: `GetMovieNextInterestingTime`,

GetTrackNextInterestingTime, and GetMediaNextInterestingTime. The normal method for stepping forward to the next frame is to use one of these calls to locate the time of the next visual sample. This works well for most media types, including video and text. Unfortunately, it does not work well for MPEG. QuickTime stores an entire MPEG stream as a single sample. Therefore stepping to the next sample would actually skip to the end of the entire sequence. To solve this problem, QuickTime 2.1 introduced a new flag, nextTimeStep, for the interesting time calls. This flag is specifically intended for stepping through frames.

The standard QuickTime movie controller has been updated to use this flag in place of the old nextTimeSample flag. The nextTimeStep flag works correctly for all media types including video and MPEG. To work correctly with MPEG, applications that implement stepping functionality should use this new flag.

See *Inside Macintosh: QuickTime* for more information on interesting times.

MediaTrackReferencesChanged

The MediaTrackReferencesChanged function notifies the derived media handler whenever the track references in the movie change.

```
pascal ComponentResult MediaTrackReferencesChanged (MediaHandler mh)
```

mh — Identifies the Movie Toolbox's connection to your derived media handler.

DISCUSSION

When an application creates, modifies, or deletes a track reference, the media handler's MediaTrackReferencesChanged function is called. When this function is called, a media handler should rebuild all information about track references and reset its values for all media inputs to their default values.

MediaTrackPropertyAtomChanged

The `MediaTrackPropertyAtomChanged` function notifies the derived media handler whenever its media property atom has changed.

```
pascal ComponentResult MediaTrackPropertyAtomChanged (MediaHandler mh)
```

`mh` Identifies the Movie Toolbox's connection to your derived media handler.

DISCUSSION

The `MediaTrackPropertyAtomChanged` function is called whenever `SetMediaPropertyAtom` is called. If the media handler uses information from the property atom, it should rebuild the information at this time.

MediaSetTrackInputMapReference

When an application modifies the media input map, the `MediaSetTrackInputMap` function provides the derived media handler with the updated input map.

```
pascal ComponentResult MediaSetTrackInputMapReference(
                    MediaHandler mh,
                    QTAtomContainer inputMap)
```

`mh` Identifies the Movie Toolbox's connection to your derived media handler.

`inputMap` Specifies the media input map for this operation. Do not modify or dispose of the input map provided.

DISCUSSION

When the `MediaSetTrackInputMapReference` function is called, the media handler should store the updated input map and recheck the types of all inputs, if it is caching this information. The input map reference passed to this function should not be disposed of or modified by the media handler.

MediaGetSampleDataPointer

The `MediaGetSampleDataPointer` function allows a derived media handler to obtain a pointer to the sample data for a particular sample number, the size of that sample, and the index of the sample description associated with that sample.

```
pascal ComponentResult MediaGetSampleDataPointer(
                    MediaHandler mh,
                    long sampleNum,
                    Ptr *dataPtr,
                    long *dataSize,
                    long *sampleDescIndex)
```

`mh` Identifies the Movie Toolbox's connection to your derived media handler.

`sampleNum` Contains the number of the sample that is to be loaded.

`dataPtr` Contains a pointer to a pointer to receive the address of the loaded sample data.

`dataSize` Contains a pointer to a field that is to receive the size, in bytes, of the sample.

`sampleDescIndex`
Contains a pointer to a long integer. The `MediaGetSampleDataPointer` function returns an index value to the sample description that corresponds to the returned sample data. If you do not want this information, set this parameter to `nil`.

DISCUSSION

The `MediaGetSampleDataPointer` function returns a pointer to the data for a particular sample number from a movie data file.

This function provides access to the base media handler's caching services for sample data. It is a service provided by the base media handler for its clients.

Each call to `MediaGetSampleDataPointer` must be balanced by a call to `MediaReleaseSampleDataPointer` or the memory will not be released.

MediaGetSampleDataPointer generally provides better overall performance than GetMediaSample.

MediaReleaseSampleDataPointer

The MediaReleaseSampleDataPointer function balances calls to MediaGetSampleDataPointer to release the memory. This function should only be used by derived media handlers.

```
pascal ComponentResult MediaReleaseSampleDataPointer(
                    MediaHandler mh,
                    long sampleNum)
```

mh — Identifies the Movie Toolbox's connection to your derived media handler.

sampleNum — Contains the number of the sample that is to be released.

MediaCompare

The MediaCompare function allows a media handler to determine whether the Movie Toolbox should allow one track to be pasted into another. MediaCompare is provided with a reference to the media with which it should be compared.

```
pascal ComponentResult MediaCompare(
                    MediaHandler mh,
                    Boolean *isOK,
                    Media srcMedia,
                    ComponentInstance srcMediaComponent)
```

mh — Identifies the Movie Toolbox's connection to your derived media handler.

isOK	Contains a pointer to a Boolean value. Your media handler must set this Boolean value to indicate whether the source media and the media associated with the media handler have equivalent media settings, so that pasting the two together would cause no media information loss.
srcMedia	Specifies the source media for this operation.
srcMediaComponent	Specifies the source media component for this operation.

MediaSetVideoParam

The MediaSetVideoParam function enables you to dynamically adjust the brightness, contrast, hue, sharpness, saturation, black level, and white level of a video image.

```
pascal ComponentResult MediaSetVideoParam(
                    MediaHandler mh,
                    long whichParam,
                    unsigned short *value)
```

mh	Identifies the Movie Toolbox's connection to your derived media handler.
whichParam	Contains a long integer which specifies the number of the video parameter that should be adjusted.
value	Contains the actual value of the video parameter. The meaning of the values vary depending on the implementation.

DISCUSSION

The MediaSetVideoParam and MediaGetVideoParam functions are currently used by the MPEG media handler.

See *Media Video Parameters* in this chapter for the constants and values you can use for the whichParam and value parameters.

MediaGetVideoParam

The `MediaGetVideoParam` function enables you to retrieve the value of the brightness, contrast, hue, sharpness, saturation, black level, or white level of a video image.

```
pascal ComponentResult MediaGetVideoParam(
                    MediaHandler mh,
                    long whichParam,
                    unsigned short *value)
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`whichParam`	Contains a long integer which specifies the number of the video parameter whose value you want to retrieve.
`value`	Contains the actual value of the requested video parameter. The meaning of the values vary depending on the implementation.

DISCUSSION

The `MediaSetVideoParam` and `MediaGetVideoParam` functions are currently used by the MPEG media handler.

See the *Inside Macintosh: QuickTime Components* chapter on Video Digitizer Components for more information about the `whichParam` and `value` parameters.

MediaSetNonPrimarySourceData

The `MediaSetNonPrimarySourceData` function allows a media handler to support receiving media data from other media handlers.

```
pascal ComponentResult MediaSetNonPrimarySourceData(
                    MediaHandler mh,
                    long inputIndex,
                    long dataDescriptionSeed,
                    Handle dataDescription,
                    void *data,
                    long dataSize,
```

```
                    ICMCompletionProcRecordPtr asyncCompletionProc,
                    UniversalProcPtr transferProc,
                    void *refCon)
```

`mh` Identifies the Movie Toolbox's connection to your derived media handler.

`inputIndex` This value is the ID of the entry in the media's input map that the data provided by the call corresponds to.

`dataDescriptionSeed`
This value is changed each time the `dataDescription` has changed. This allows for a quick check by the media handler to see if the `dataDescription` has changed.

`dataDescription`
A handle to a data structure describing the input data.

`data` Points to the input data. This pointer must contain a 32-bit clean address.

`dataSize` Contains the size of the sample in bytes.

`asyncCompletionProc`
Points to a completion function structure. If the `asyncCompletionProc` is set to `nil`, the data pointer will only be valid for the duration of this call. If the `asyncCompletionProc` is not `nil`, it contains an `ICMCompletionProcRecord` that must be called when your media handler is done with the provided data pointer.

`transferProc` A routine that allows the application to transform the type of the input data to the kind of data preferred by the codec. The client of the codec passes the source data in the form most convenient for it. If the codec needs the data in another form, it can negotiate with the client or directly with the Image Compression Manager to obtain the required data format.

`refCon` Contains a reference constant (defined as a void pointer). Your application specifies the value of this reference constant in the function structure you pass to the media handler.

DISCUSSION

There are two parts to supporting modifier tracks in a derived media handler: sending and receiving. The base media handler takes care of sending data for all its clients. Therefore, authors of derived media handlers do not usually need to implement sending data support.

Receiving data is a more complex situation. The base media handler takes care of input types that it understands. The base media handler supports the following types of data:

```
kTrackModifierTypeMatrix
kTrackModifierTypeGraphicsMode
kTrackModifierTypeClip
kTrackModifierTypeVolume
kTrackModifierTypeBalance
```

If a media handler wants to support receiving other types of data it must implement the `MediaSetNonPrimarySourceData` routine. `MediaSetNonPrimarySourceData` is called by modified tracks to supply the current data for each input. All unrecognized input types should be delegated to the base media handler so that they can be handled.

The following is a basic shell implementation of a derived media handler's `MediaSetNonPrimarySourceData` function. Note that it is necessary to delegate all unhandled input types to the base media handler.

```
pascal ComponentResult MySetNonPrimarySourceData( MyGlobals store,
    long inputIndex, long dataDescriptionSeed, Handle dataDescription,
    void *data, long dataSize,
    ICMCompletionProcRecordPtr asyncCompletionProc,
    UniversalProcPtr transferProc, void *refCon )
{
ComponentResult err = noErr;
QTAtom inputAtom;
QTAtom typesAtom;
long inputType;

// determine what kind of input this is
inputAtom = QTFindChildByID(store->inputMap,
    kParentAtomIsContainer, kTrackModifierInput, inputIndex, nil);
if (!inputAtom) {
    err = cannotFindAtomErr;
```

```
    goto bail;
}
    typesAtom = QTFindChildByID(store->inputMap, inputAtom,
    kTrackModifierType, 1, nil);
err = QTCopyAtomDataToPtr(store->inputMap, typesAtom, false,
    sizeof(inputType), &inputType, nil);
if (err) goto bail;

switch(inputType) {
    case kMyInputType:
        if (data == nil) {
            // no data, reset to default value
        }
        else {
            // use this data
            // when done, notify caller we're done with this data
            if (asyncCompletionProc)
                CallICMCompletionProc(
                    asyncCompletionProc->completionProc,
                    noErr, codecCompletionSource | codecCompletionDest,
                    asyncCompletionProc->completionRefCon);
        }
        break;

    default:
        err = MediaSetNonPrimarySourceData(store->delegateComponent,
            inputIndex, dataDescriptionSeed, dataDescription, data,
            dataSize, asyncCompletionProc, transferProc, refCon);
        break;
}
bail:
    return err;
}
```

MediaGetOffscreenBufferSize

The `MediaGetOffscreenBufferSize` function determines the dimensions of the offscreen buffer.

```
pascal ComponentResult MediaGetOffscreenBufferSize(
                    MediaHandler mh,
                    Rect *bounds,
                    short depth,
                    CTabHandle ctab)
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`bounds`	Specifies the boundaries of your offscreen buffer.
`depth`	Specifies the depth of the offscreen.
`ctab`	Contains a handle to the color table associated with the offscreen buffer.

DISCUSSION

Before the base media handler allocates an offscreen buffer for your derived media handler, it calls your `MediaGetOffscreenBufferSize` function. The depth and color table used for the buffer are also passed. When this function is called, the `bounds` parameter specifies the size that the base media handler intends to use for your offscreen buffer. You can modify this as appropriate before returning. This capability is useful if your media handler can draw only at particular sizes. It is also useful for implementing anti-aliased drawing (you can request a buffer that is larger than your destination area and have the base media handler scale the image down for you).

RESULT CODES

badComponentInstance	0x80008001	Invalid component instance specified

MediaSetHints

The MediaSetHints function implements the appropriate behavior for the various media hints such as scrub mode and high-quality mode.

```
pascal ComponentResult MediaSetHints(
                    MediaHandler mh,
                    long hints)
```

mh	Identifies the Movie Toolbox's connection to your derived media handler.
hints	Contains all hint bits that currently apply to the given media.

DISCUSSION

When an application calls SetMoviePlayHints or SetMediaPlayHints, your media handler's MediaSetHints routine is called.

RESULT CODES

badComponentInstance	0x80008001	Invalid component instance specified

MediaGetName

The MediaGetName function returns the name of the media type. For example, the video media handler returns the string "video."

```
pascal ComponentResult MediaGetName(
                    MediaHandler mh,
                    Str255 name,
                    long requestedLanguage,
                    long *actualLanguage)
```

`mh` Identifies the Movie Toolbox's connection to your derived media handler.

`name` Specifies where to return the name of the media type.

`requestedLanguage`
Specifies the language in which you want the `name` returned. This value is a standard Mac OS region code.

`actualLanguage`
Specifies the actual language in which the `name` is returned. This value is a standard Mac OS region code.

function result The name of the media type.

RESULT CODES

`badComponentInstance`	0x80008001	Invalid component instance specified

Graphics Data Management

This section describes functions for managing graphics data. These functions apply to all derived media handler components.

MediaGetDrawingRgn

The `MediaGetDrawingRgn` function allows your derived media handler component to specify a portion of the screen that must be redrawn. This region is defined in the movie's display coordinate system.

```
pascal ComponentResult MediaGetDrawingRgn (ComponentInstance ci,
                    RgnHandle *partialRgn);
```

`ci` Identifies the Movie Toolbox's connection to your derived media handler.

`partialRgn` Points to a handle that defines the screen region to be redrawn. Note that your component is responsible for disposing of this region once drawing is complete. Since the base media handler will use this region during redrawing, it is best to dispose of it when your component is closed.

DISCUSSION

The Movie Toolbox calls this function in order to determine what part of the screen needs to be redrawn. By default, the Movie Toolbox redraws the entire region that belongs to your component. If your component determines that only a portion of the screen has changed, and has indicated this to the Movie Toolbox by setting the `mPartialDraw` flag to 1 in the `flagsOut` parameter of the `MediaIdle` function, the Movie Toolbox calls your component's `MediaGetDrawingRgn` function. Your component returns a region that defines the changed portion of the track's display region.

RESULT CODES

`badComponentSelector`	0x80008002	Function not supported

Memory Manager errors

MediaGetGraphicsMode

The `MediaGetGraphicsMode` function allows you to obtain the graphics mode and blend color values currently in use by any media handler.

```
pascal HandlerError MediaGetGraphicsMode (
                    MediaHandler mh,
                    long *mode,
                    RGBColor *opColor);
```

`mh` Identifies the Movie Toolbox's connection to your derived media handler.

`mode` Contains a pointer to a long integer. The media handler returns the graphics mode currently in use by the media handler. This is a QuickDraw transfer mode value.

opColor — Contains a pointer to an RGB color structure. The Movie Toolbox returns the color currently in use by the media handler. This is the blend value for blends and the transparent color for transparent operations. The Movie Toolbox supplies this value to QuickDraw when you draw in addPin, subPin, blend, transparent, or graphicsModeStraightAlphaBlend mode.

RESULT CODES

Component Manager errors, as documented in *Inside Macintosh: More Macintosh Toobox*.

SEE ALSO

You can set the graphics mode and blend color of any media handler by calling the MediaSetGraphicsMode function, which is described in the next section.

MediaSetGraphicsMode

The MediaSetGraphicsMode function allows you to set the graphics mode and blend color of any media handler.

```
pascal HandlerError MediaSetGraphicsMode (
                    MediaHandler mh,
                    long mode,
                    const RGBColor *opColor);
```

mh — Identifies the Movie Toolbox's connection to your derived media handler.

mode — Specifies the graphics mode of the media handler. This is a QuickDraw transfer mode value.

opColor — Contains a pointer to the color for use in blending and transparent operations. The media handler passes this color to QuickDraw as appropriate when you draw in addPin, subPin, blend, transparent, or graphicsModeStraightAlphaBlend mode.

RESULT CODES

Component Manager errors, as documented in *Inside Macintosh: More Macintosh Toobox*.

SEE ALSO

You can retrieve the graphics mode and blend color currently in use by any media handler by calling the `MediaGetGraphicsMode` function, which is described in the previous section.

Sound Data Management

This section describes functions for managing sound data. These functions apply to all derived media handler components.

MediaSetSoundLocalizationData

If you are creating a media handler that plays sound and wish to support 3D Sound capabilities, you need to implement the `MediaSetSoundLocalizationData` routine.

```
pascal ComponentResult MediaSetSoundLocalizationData (MediaHandler mh,
                    Handle data)
```

`mh` Identifies the Movie Toolbox's connection to your derived media handler.

`data` The data passed to your media handler, in the format of a Sound Sprockets `SSpLocalizationData` record.

DISCUSSION

This routine is passed a handle containing the new `SSpLocalizationData` record to use. If the handle is `nil`, it indicates that no 3D Sound effects should be used. If you implement this routine, and return `noErr` as the result, it is assumed that your media handle assumes responsibility for disposing of the data handle passed. If the implementation of this routine returns an error, the caller will dispose of the handle. This behavior is implemented to minimize the copying

of the settings handle, making it easier for developers to implement the `MediaSetSoundLocalizationData` function.

Note: The `MediaSetSoundLocalizationData` will be called regardless of whether the 3D Sound settings were set on the track using `SetTrackSoundLocalizationSettings` or via the Modifier Track mechanism.

Tween Media Handler Components

Contents

This chapter describes the tween media handler and a set of functions you can implement to create a tween component. The tween media handler and tween components were introduced in QuickTime 2.5.

This chapter is divided into the following major sections:

- "About the Tween Media Handler" introduces the tween media handler and tween components.
- "Using the Tween Media Handler" describes how you can use existing tween components with the tween media handler and how to create your own tween components.
- "Tween Media Handler Reference" describes the constants, data types, and functions for use with the tween media handler and tween components.

About the Tween Media Handler

The **tween media handler** is a track that is used exclusively as a modifier track. It is used to algorithmically generate values to modify the playback of other tracks. The tween media handler sends values to other media handlers; it never presents data. For information about modifier tracks, see the chapter "Movie Toolbox" in this book.

Using a tween media track as a modifier track differs from using a base media track in that the tween media handler only requires a start value and a stop value to be specified, whereas, a base media handler requires a series of discrete values to be specified.

The value sent to another track is the start value plus the difference between the stop value and the start value multiplied by the elapsed percentage of the duration of the sample. Thus, the value sent halfway through the sample is the start value plus the difference between the stop and start value multiplied by .50. The formula is as follows:

$$value = start + \langle (stop - start) \cdot percent \rangle$$

This kind of calculation is called **interpolation**. Specifically, it is a linear interpolation because the range of possible values between the start and stop values can be graphed as a straight line.

The interpolation provided by the tween media handler is implemented by tween components. There is one component for each kind of data from which the values are derived. The Apple-supplied components support the following kinds of data:

- long and short integers
- fixed-point numbers
- QuickDraw points and rectangles
- RGB colors
- QuickTime 3x3 matricies
- QuickDraw 3D transforms, scales, rotations, matricies, quaternions, and cameras
- 3D sound localization data

For a complete list of Apple-supplied tween components, see "Constants" (page 13-12).

Apple-supplied components only support linear interpolation for deriving tween values. You can implement your own components to support other data types or to perform non-linear interpolation.

Using the Tween Media Handler

You use the tween media handler to send tween values from a tween media track to a receiving track, such as a video track or a sound track. To send tween values, you must create a tween media track. The tween media handler sends the values for you based on how you set up the sample data in the track. You may also decide to implement a tween component from which you can derive tween values.

This section is divided into the following topics:

- "Creating a Tween Track" discusses how to create a tween media track that modifies a receiving track
- "Creating a Tween Component" describes the steps you must take to implement your own tween component

Creating a Tween Track

To create a tween media track, you must:

- Create a tween track and its media.
- Create one or more tween media samples.
- Add the media samples to the tween media.
- Add the tween media to the track.
- Create a link from the tween track to the track to which the tween media handler should send tween values.
- Bind the tween entry to the desired attributes in the receiving track.

The sample code shown in this section creates a tween sample that interpolates a short integer from 512 to 0. The tween media is attached to the sound track of a QuickTime movie to modify the sound track's volume. The data type for the tween component is `kTweenTypeShort`.

The sample code shown in Listing 13-1 creates a new track (`t`) to be used as the tween track and new tween media (type `TweenMediaType`).

Listing 13-1 Creating a tween track and tween media

```
Track t;
Media md;
SampleDescriptionHandle desc;

// ...
// set up the movie, m
// ...

// allocate a sample description handle
desc = (SampleDescriptionHandle)NewHandleClear (
    sizeof (SampleDescription));

// create the tween track, t
t = NewMovieTrack (m, 0, 0, kNoVolume);

// create the tween media, md
```

```
md = NewTrackMedia (t, TweenMediaType, 600, nil, 0);

(**desc).descSize = sizeof(SampleDescription);
```

Next, an application must create a tween media sample. The tween media sample is defined as a QT atom container structure that contains one or more `kTweenEntry` atoms. (See the chapter “MovieToolBox” for additional information on QT atom containers.) Each `kTweenEntry` atom defines a separate tween operation. A single tween sample can describe several parallel tween operations.

The sample code shown in Listing 13-2 creates a new QT atom container and inserts a `kTweenEntry` atom into the container. Then, it creates two leaf atoms, both children of the `kTweenEntry` atom. The first leaf atom (atom type `kTweenType`) contains the type of the tween data, `kTweenTypeShort`. The second leaf atom (atom type `kTweenData`) contains the two data values for the tween operation, 512 and 0.

Listing 13-2 Creating a tween sample

```
QTAtomContainer container = nil;
short tweenDataShort[2];
QTAtomType tweenType;

tweenDataShort[0] = 512;
tweenDataShort[1] = 0;

// create a new atom container to hold the sample
QTNewAtomContainer (&container);

// create the parent tween entry atom
tweenType = kTweenTypeShort;
QTInsertChild (container, kParentAtomIsContainer, kTweenEntry, 1, 0, 0,
    nil, &tweenAtom);

// add two child atoms to the tween entry atom
// * the type atom, kTweenType
QTInsertChild (container, tweenAtom, kTweenType, 1, 0,
    sizeof(tweenType), &tweenType, nil);
```

```
// * the data atom, kTweenData
QTInsertChild (container, tweenAtom, kTweenData, 1, 0, sizeof(short) * 2,
    tweenDataShort, nil);
```

You do not have to start the tween at the beginning of the sample, nor do you have to stop at the end of the sample. You can specify the start of the tween and its duration by adding additional child atoms to the tween entry. You can add a `kTweenStartOffset` atom to start the tween operation at 500 units into the sample with the following lines of code:

```
TimeValue time = 500;
QTInsertChild (container, tweenAtom, kTweenStartOffset, 1, 0,
    sizeof(TimeValue), &time, nil);
```

You can specify a duration for the tween operation independent of the sample's duration by adding a `kTweenDuration` atom to the tween entry, as follows:

```
TimeValue duration = 1000;
QTInsertChild (container, tweenAtom, kTweenDuration, 1, 0,
    sizeof(TimeValue), &duration, nil);
```

Once the tween samples have been created, you can add them to the tween media and then add the tween media to the track, as shown in Listing 13-3.

Listing 13-3 Adding the tween sample to the media and the media to the track

```
// add the sample to the tween media
BeginMediaEdits (md);
AddMediaSample (md, container, 0,
    GetHandleSize(container), kSampleDuration, desc, 1, 0, nil);
EndMediaEdits(md);

// dispose of the sample description handle and the atom container
DisposeHandle ((Handle)desc);
QTDisposeAtomContainer(container);

// add the media to the track
InsertMediaIntoTrack(t, 0, 0, kSampleDuration, kFix1);
```

Once you have added the tween media to its track, you need to call the `AddTrackReference` function to create a link between the tween track to the receiving track. `AddTrackReference` returns the index of the reference it creates.

The sample code shown in Listing 13-4 retrieves the sound track from a movie and calls `AddTrackReference` to create a link between the tween track (`t`) and the sound track. The reference index is returned in the parameter `referenceIndex`.

Listing 13-4 Creating a link between the tween track and the sound track

```
Track soundTrack;

// retrieve the sound track from the movie
soundTrack = GetMovieIndTrackType (theMovie , 1,
    AudioMediaCharacteristic,
    movieTrackCharacteristic | movieTrackEnabledOnly);

long referenceIndex;

// create a link between the tween track and the sound track
// on return, referenceIndex contains the index of the link
err = AddTrackReference (soundTrack, t, kTrackModifierReference,
    &referenceIndex);
```

Once you have linked the tween track to its receiving track, you must update the input map of the receiving track's media to indicate how the receiving track should interpret the data it receives from the tween track. To do this, you create a QT atom container and insert an atom of type `kTrackModifierInput` whose ID is the index returned by the `AddTrackReference` function. Then, you insert two atoms as children of the `kTrackModifierInput` atom:

- A leaf atom of type `kTrackModifierType` that contains the attribute of the receiving track to be modified. For example, if the tween entry modifies the matrix of the track, the leaf atom would contain the type `kTrackModifierTypeMatrix`.
- A leaf atom of type `kInputMapSubInputID` that contains the ID of the tween entry atom. This binds the tween entry to the receiving track.

Once you have created the appropriate atoms in the input map, you call `SetMediaInputMap` to assign the input map to the receiving track's media.

The sample code shown in Listing 13-5 creates an input map for the sound track of a movie. In this code, the tween media is linked to a sound track; the interpolated tween values are used to modify the sound track's volume.

Listing 13-5 Binding a tween entry to its receiving track

```
QTAtomContainer inputMap = nil;

// create an atom container to hold the input map
if (QTNewAtomContainer (&inputMap) == noErr)
{
    QTAtom inputAtom;
    OSType inputType;
    long tweenID = 1;

    // create a kTrackModifierInput atom
    // whose ID is referenceIndex
    QTInsertChild(inputMap, kParentAtomIsContainer,
        kTrackModifierInput, referenceIndex, 0, 0, nil,
        &inputAtom);

    // add a child atom of type kTrackModifierTypeVolume
    inputType = kTrackModifierTypeVolume;
    QTInsertChild (inputMap, inputAtom, kTrackModifierType, 1, 0,
       sizeof(inputType), &inputType, nil);

    // add a child atom for the ID of the tween to
    // modify the volume
    QTInsertChild (inputMap, inputAtom, kInputMapSubInputID, 1,
        0, sizeof(tweenID), &tweenID, nil);

    // assign the input map to the sound media
    SetMediaInputMap(GetTrackMedia(soundTrack), inputMap);

    // dispose of the input map
    QTDisposeAtomContainer(inputMap);
}
```

Creating a Tween Component

Your tween component must provide three functions in addition to the standard functions required to implement a component. The functions you must provide are `TweenInitialize`, `TweenDoTween`, and `TweenReset`. The following examples show a tween component that interpolates values for short integers. QuickTime provides a component for short integers (`kTweenTypeShort`) for you; you do not need to implement a component to handle interpolation of short integers yourself.

Listing 13-6 shows the `TweenerShortInitialize` function, which QuickTime calls to set up the component. In this example, `TweenerShortInitialize` simply returns. In a more complex example, `TweenerShortInitialize` might allocate storage to be used during the tween operation.

Listing 13-6 A function to initialize a tween component

```
pascal ComponentResult TweenerShortInitialize(TweenerComponent tc,
                                               QTAtomContainer container,
                                               QTAtom tweenAtom,
                                               QTAtom dataAtom)
{
    return noErr;
}
```

Listing 13-7 shows the `TweenShortDoTween` function, which QuickTime calls when it needs to send a tween value from the tween track to a media track. The data atom and the function that is called to set the value are stored in the tween record.

Listing 13-7 A function to set a value during a tween operation

```
pascal ComponentResult TweenShortDoTween(TweenerComponent tc,
                                         TweenRecord *tr)
{
    short *data;
    short tFrom, tTo, tValue;

    QTGetAtomDataPtr(tr->container, tr->dataAtom, nil, (Ptr *)&data);
```

```
    tFrom = data[0];
    tTo = data[1];
    tValue = tFrom + FixMul(tTo - tFrom, tr->percent);

    (tr->dataProc)((struct TweenRecord *)tr, &tValue,
        sizeof(tValue), 1, nil, nil, nil, nil);

    return noErr;
}
```

Listing 13-8 shows the `TweenShortReset` function, which QuickTime calls after the tween sample completes. In this example, because `TweenerShortInitialize` does not allocate any storage, `TweenerShortReset` simply returns. In a more complex example, `TweenerShortReset` would release any storage allocated by `TweenerShortInitialize` and any storage allocated during the tween operation.

Listing 13-8 A function to reset a tween component

```
pascal ComponentResult TweenerShortReset (TweenerComponent tc)
{
    return noErr;
}
```

Tween Media Handler Reference

This section describes the constants, data types, and routines associated with the tween media handler and tween components.

Constants

Tween Component Constant

The `TweenComponentType` constant specifies that the component is a tween component.

```
enum {
    TweenComponentType  = 'twen'
};
```

Tween Atom Types

The following atom types are defined for tween-related atoms:

```
enum {
    kTweenEntry                 = 'twen',
    kTweenData                  = 'data',
    kTweenType                  = 'twnt',
    kTweenStartOffset           = 'twst',
    kTweenDuration              = 'twdu',
    kTween3dInitialCondition    = 'icnd',
    kTweenInterpolationStyle    = 'isty',
    kTweenRegionData            = 'qdrg',
    kTweenPictureData           = 'PICT'
};

    kInputMapSubInputID         = 'subi',
};
```

`kTweenEntry` A parent atom that defines a set of values for a tween operation. Its child atoms define the tween data type, the tween data, and additional attributes of the tween operation.

`kTweenData` A leaf atom that contains the data values for a tween operation. You create a `kTweenData` atom as a child of a `kTweenEntry` atom.

`kTweenType` A leaf atom that contains the type of the data for a tween operation (for example, `kTweenTypeShort`). You create a `kTweenType` atom as a child of a `kTweenEntry` atom.

`kTweenStartOffset` A leaf atom that contains the starting offset, in time units, of a tween operation. You create a `kTweenStartOffset` atom as a child of a `kTweenEntry` atom. (Optional)

`kTweenDuration` A leaf atom that contains the duration, in time units, of a tween operation. You create a `kTweenDuration` atom as a child of a `kTweenEntry` atom. (Optional)

`kTween3dInitialCondition`
A leaf atom that contains an initial value for a 3D tween operation. You create a `kTween3DInitialCondition` atom as a child of a `kTweenEntry` atom. The type of the initial value should be the same as the type of data to be tweened. If you do not provide an initial value, the tween operation is performed against a value of 0. (Optional)

`kTweenInterpolationStyle`
Reserved.

`kTweenRegionData` A leaf atom that contains a region. You create a `kTweenRegionData` atom as an additional child of the `kTweenEntry` atom when the tween type is `kTweenTypeQDRegion`. The region is mapped onto the interpolated rectangle calculated by the tween operation.

`kTweenPictureData` A leaf atom that contains a PICT image. You create a `kTweenPictureData` atom as a child of a `kTweenEntry` atom. Only used when tween type is between Region Data.

Media Input Map

The following input type is defined for tween-related atoms:

```
enum {
    kInputMapSubInputID         = 'subi',
};
```

`kInputMapSubInputID`
A leaf atom that contains the ID of a tween entry. You create a `kInputMapSubInputID` atom in a receiving track's input map to define the relationship between the tween

entry and the receiving track. You create a `kInputMapSubInputID` atom as a child of a `kTrackModifierInput` atom.

Tween Data Types

Each tween component is identified by a media subtype that specifies the kind of data handled by the component. The media subtype also specifies the kind of interpolation that occurs. The data is placed in an atom of type `kTweenData`. The kind of data stored for a tween component is specified by one of the following constants:

```
enum {
    kTweenTypeShort                     = 1,
    kTweenTypeLong                      = 2,
    kTweenTypeFixed                     = 3,
    kTweenTypePoint                     = 4,
    kTweenTypeQDRect                    = 5,
    kTweenTypeQDRegion                  = 6,
    kTweenTypeMatrix                    = 7,
    kTweenTypeRGBColor                  = 8,
    kTweenTypeGraphicsModeWithRGBColor  = 9,
    kTweenType3dScale                   = '3sca',
    kTweenType3dTranslate               = '3tra',
    kTweenType3dRotate                  = '3rot',
    kTweenType3dRotateAboutPoint        = '3rap',
    kTweenType3dRotateAboutAxis         = '3rax',
    kTweenType3dQuaternion              = '3qua',
    kTweenType3dMatrix                  = '3mat',
    kTweenType3dCameraData              = '3cam',
    kTweenType3dSoundLocalizationData   = '3slc'
};
```

`kTweenTypeShort` Two signed 16-bit integers from which to interpolate the tween value.

`kTweenTypeLong` Two signed 32-bit integers from which to interpolate the tween value.

`kTweenTypeFixed` Two 32-bit fixed point values from which to interpolate the tween value.

`kTweenTypePoint` Two QuickDraw points from which to interpolate the tween value. Each coordinate (h and v) of the points is interpolated independently. The results of the interpolation on each coordinate are used to specify the tween value, which is a point.

`kTweenTypeQDRect` Two QuickDraw rectangles. Each coordinate (r, l, b, t) of the rectangles is interpolated independently. The result of the interpolation on each coordinate is used to create the tween value, which is a rectangle.

`kTweenTypeQDRegion` Two QuickDraw rectangles and a region, stored in a `kTweenRegionData` atom. Each coordinate (r, l, b, t) of the rectangles is interpolated independently. The result of the interpolation on each coordinate specify a rectangle, into which the region is mapped. The resultant region specifies the tween value.

Alternatly, the `kTweenPictureData` atom can be used instead of `kTweenTypeQDRegion`. The picture is used to generate a region by imaging it the size specified by interpolaring the two rectangles. This allows for smoother region based tween operations to be created.

`kTweenTypeMatrix` Two QuickTime 3x3 matrices. The corresponding cells in the matricies are interpolated independently. The result of the interpolation on each cell specifies a tween value, which is a matrix. Only matrices that specify translation and scaling should be used.

`kTweenTypeRGBColor` Two RGB colors from which to interpolate the tween value. Each color component (r, g, b) of the colors is interpolated independently. The results of the interpolation on each color component are used to specify the tween value, which is an RGB color.

`kTweenTypeGraphicsModeWithRGBColor`
Two `ModifierTrackGraphicsModeRecord` records from which to interpolate a tween value. Only the color is interpolated. The graphics mode is taken from the first modifier track graphic mode received.

`kTweenType3dScale` A QuickTime 3D `TQ3Vector3D` record, from which the scale transform is used to interpolate a tween value.

`kTweenType3dTranslate`
A QuickTime 3D `TQ3Vector3D` record, from which the translation transform is used to interpolate a tween value.

`kTweenType3dRotate` A QuickTime 3D `TQ3RotateTransformData` record, from which the rotation transform is used to interpolate a tween value.

`kTweenType3dRotateAboutPoint`
A QuickTime 3D `TQ3RotateAboutPointTransformData` record, from which the rotate about point transform is used to interpolate a tween value.

`kTweenType3dRotateAboutAxis`
A QuickTime 3D `TQ3RotateAboutAxisTransformData` record, from which the angle field of the rotate about axis transform is used to interpolate a tween value.

`kTweenType3dQuaternion`
A QuickTime 3D `TQ3Quaternion` record, from which the quaternion transform is used to interpolate a tween value.

`kTweenType3dMatrix` A QuickTime 3D `TQ3Matrix4x4` record, from which the 4x4 matrix transform is used to interpolate a tween value.

`kTweenType3dCameraData`
A QuickTime 3D `TQ3CameraData` record, from which the camera data is used to interpolate a tween value.

`kTweenType3dSoundLocalizationData`
Two `SSpLocalizationData` records, from which the sound data is used to interpolate a 3D sound value.

Data Types

The following sections describe the component instance definition, tween record, and the value setting prototype function used by tween components.

Component Instance

The component instance for a component, `TweenerComponent`, identifies an application's use of a component. For more information about component instances, see the Component Manager chapter of *Inside Macintosh: More Toolbox Essentials*.

```
typedef ComponentInstance TweenerComponent;
```

Tween Record

QuickTime maintains a tween record structure that is provided to your tween component's `TweenDoTween` method. The `TweenRecord` structure is defined as follows.

```
typedef struct TweenRecord TweenRecord;

struct TweenRecord {
    long                    version;
    QTAtomContainer         container;
    QTAtom                  tweenAtom;
    QTAtom                  dataAtom;
    Fixed                   percent;
    TweenerDataUPP          dataProc;
    void *                  private1;
    void *                  private2;
};
```

Field descriptions

`version`	The version number of this structure. This field is initialized to 0.
`container`	The atom container that contains the tween data.
`tweenAtom`	The atom for this tween entry's data in the container.
`percent`	The percentage by which to change the data.
`dataProc`	The procedure the tween component calls to send the tweened value to the receiving track.
`private1`	Reserved.
`private2`	Reserved.

Value Setting Function

The function that you call to send the interpolated value to the receiving track is defined as a universal procedure in systems that support the Code Fragment Manager (CFM) or defined as a data procedure for non-CFM systems:

```
typedef UniversalProcPtr TweenerDataUPP;    /* CFM */

typedef TweenerDataProcPtr TweenerDataUPP;  /* non-CFM */
```

The `TweenerDataUPP` function pointer specifies the function the tween component calls with the value generated by the tween operation. A tween component calls this function from its implementation of the `TweenerDoTween` function.

```
typedef pascal ComponentResult (*TweenerDataProcPtr)(
                    TweenRecord *tr,
                    void *tweenData,
                    long tweenDataSize,
                    long dataDescriptionSeed,
                    Handle dataDescription,
                    ICMCompletionProcRecordPtr asyncCompletionProc,
                    ProcPtr transferProc,
                    void *refCon);
```

`tr` Contains a pointer to the tween record for the tween operation.

`tweenData` Contains a pointer to the generated tween value.

`tweenDataSize` Specifies the size, in bytes, of the tween value.

`dataDescriptionSeed`
Specifies the starting value for the calculation. Every time the content of the `dataDescription` handle changes, this value should be incremented.

`dataDescription`
Specifies a handle containing a description of the tween value passed. For basic types such as integers, the calling tween component should set this parameter to `nil`. For more complex types such as compressed image data, the calling tween component should set this handle to contain a description of the tween value, such as an image description.

`asyncCompletionProc`
Contains a pointer to a completion procedure for asynchronous operations. The calling tween component should set the value of this parameter to `nil`.

transferProc	Contains a pointer to a procedure to transfer the data. The calling tween component should set the value of this parameter to nil.
refCon	Contains a pointer to a reference constant. The calling tween component should set the value of this parameter to nil.

DISCUSSION

You call this function by invoking the function specified in the tween record's dataProc field.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified

Tween Component Functions

This section describes the functions that you must provide to implement a tween component. QuickTime calls these functions when it uses your component.

TweenerInitialize

The TweenerInitialize function is called to initialize your tween component for a single tween operation.

```
extern pascal ComponentResult TweenerInitialize(
                    TweenerComponent tc,
                    QTAtomContainer container,
                    QTAtom tweenAtom,
                    QTAtom dataAtom)
```

tc	Specifies the tween component for this operation.
container	Specifies the container that holds the atoms specified by the tweenAtom and dataAtom parameters.

tweenAtom	Specifies the atom that contains all parameters for defining this tween. This includes the data atom and any special atoms, such as an atom of type kTweenRegionData, that may be necessary.
dataAtom	Specifies the atom that contains the values to be tweened. This atom is a child of the atom specified by the tweenAtom parameter.

DISCUSSION

This function sets up the tween component when it is first used. In your TweenerInitialize function, you can allocate storage and set up any structures that you need for the duration of a tween operation. Although the container that holds the data atom is available during each call to the TweenerDoTween function, you can improve the performance of your tween component by extracting the data to be used by the TweenerDoTween function in the TweenerInitialize function.

The data atom parameter is provided as a convenience; you can also call QT atom container functions to locate the data atom in the container. For more information about the QT atom container functions, see the chapter "Movie Toolbox" in this book.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified

TweenerReset

The TweenerReset function is called to clean up when the tween operation is finished.

```
extern pascal ComponentResult TweenerReset (TweenerComponent tc)
```

tc	Specifies the tween component for this operation.

DISCUSSION

This function releases storage allocated by the tween component when the component is no longer being used. The `TweenerReset` function should release any storage allocated by the `TweenerInitialize` function and close or release any other resources used by the component. A tween component may receive a tweener initialize or a close call after being reset.

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified

TweenerDoTween

The `TweenerDoTween` function is called to perform a tween operation.

```
extern pascal ComponentResult TweenerDoTween(
                    TweenerComponent tc,
                    TweenRecord *tr)
```

`tc` Specifies the tween component for this operation.

`tr` Contains a pointer to the tween record for the tween operation.

DISCUSSION

QuickTime calls this function to interpolate the data used during a tween operation. The tween record contains complete information about the tween operation, including the start and end values for the operation and a percentage that indicates the progress towards completion of the tween sample. For more information about the structure of a tween record, see "Tween Record" (page 13-17).

Your `TweenerDoTween` function should use the information in the tween record to calculate the tweened value. `TweenerDoTween` should call the data function specified in the tween record, passing it the tweened value. For more information about the data procedure, see "Value Setting Function" (page 13-17).

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified

Sprite Media Handler

Contents

This chapter describes the sprite media handler, a media handler you can use to add a sprite animation track to a QuickTime movie. This chapter is divided into the following major sections:

- "About the Sprite Media Handler" introduces the sprite media handler and describes the characteristics of a sprite track.
- "Using the Sprite Media Handler" describes how you can create a sprite track and add it to a QuickTime movie.
- "Sprite Media Handler Reference" describes the constants, data types, and functions for use with the sprite media handler.

About the Sprite Media Handler

The sprite media handler is a media handler that makes it possible to add a track containing a sprite animation to a QuickTime movie. The sprite media handler provides routines for manipulating the sprites and images in a sprite track. The sprite media handler makes use of routines provided by the Sprite Toolbox. For background information about sprites, sprite animation, and the Sprite Toolbox, see Chapter 1, "Movie Toolbox."

As with sprites created in a sprite world, sprites in a sprite track have properties that define their locations, images, and appearance. However, you create the sprite track and its sprites differently than you create the sprites in a sprite world.

A **sprite track** is defined by one or more key frame samples, each followed by any number of override samples. A key frame sample and its subsequent override samples define a scene in the sprite track. A key frame sample is a QT atom container that contains atoms defining the sprites in the scene and their initial properties. The override samples are other QT atom containers that contain atoms that modify sprite properties, thereby animating the sprites in the scene. For more information about QT atoms and atom containers, see Chapter 1, "Movie Toolbox."

A key frame sample also contains all of the images used by the sprites. This allows the sprites in a sprite track to share image data. The images consist of two parts, an image description handle (`ImageDescriptionHandle`) concatenated with a compressed image. The image description handle describes the compressed image. You can compress the image using any QuickTime codec.

Images are stored in a key frame sample by index; each sprite has an image index property (`kSpritePropertyImageIndex`) that specifies the sprite's current image. All images assigned to a sprite should be created using the same image description.

The matrix, layer, visible, and graphics mode sprite properties have the same meaning for a sprite in a sprite track as for a sprite created in a sprite world.

As with sprite worlds, you can create a sprite track that has a solid background color, a background image composed of the images of one or more background sprites, or both a background color and a background image.

Key Frame Samples and Override Samples

A sprite track is defined by one or more key frame samples, each followed by any number of override samples. A key frame sample for a sprite track defines the following aspects of a sprite track:

- The number of sprites in the scene and their initial properties.
- All of the shared image data to be used by the sprites in the scene, including image data to be used in the subsequent override samples. Because a key frame sample contains the image data for the scene, the key frame sample tends to be larger than its subsequent override samples.

An override sample overrides some aspect of the key frame sample. For example, an override sample might modify the location of sprites defined in the key frame sample. Override samples do not contain any image data, so they can be very small. An override sample can show or hide a sprite defined in the key frame sample, but it cannot define new sprites or remove sprites defined in its key frame sample. An override sample can override any number of properties for any number of sprites. For example, a single override sample might change the layer and location of sprite ID three, and hide sprite ID ten.

There are two sprite track formats that define how a key frame sample and its subsequent override samples are interpreted. If the current sample is a key frame sample, the key frame sample alone fully describes the current state of the track. If the current sample is an override sample, the current state may differ depending on the sprite track format:

- If the sprite track format is `kKeyFrameAndSingleOverride`, the current state is defined by the most recent key frame sample and the current override sample. This is the default format. The advantage of this format is that it allows for excellent performance during random access. A sprite track that

uses this format can play backwards and drop frames smoothly. The disadvantage of this format is that the file size of the track may be larger than a track that uses the other format.

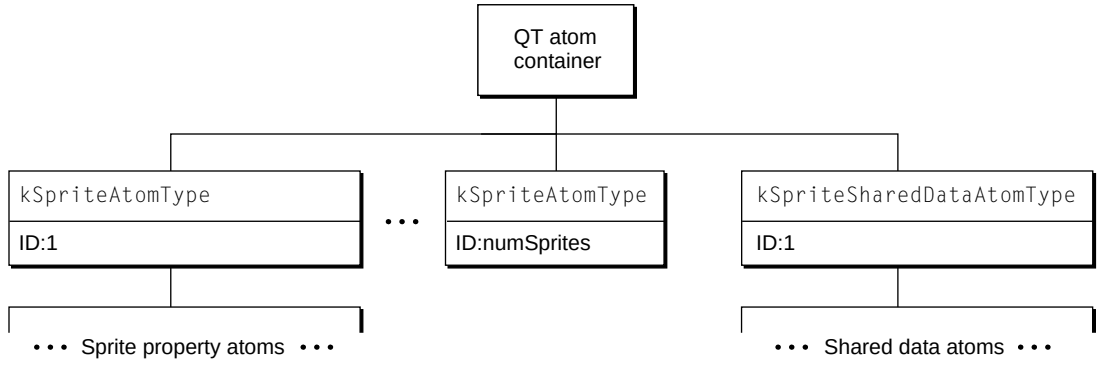
- If the sprite track format is `kKeyFrameAndAllOverrides`, the current state is defined by the most recent key sample and all subsequent override samples, including the current override sample. This format results in a smaller file size. However, you should not use this format if you want your sprite track to play backwards or drop frames smoothly. When you play a movie that contains a sprite track whose format is `kKeyFrameAndAllOverrides`, you should configure the movie to play all frames.

Sprite Track Media Format

The sprite track media format is hierarchical and is based on QT atoms and atom containers. A sprite track sample is a flattened QT atom container structure. A new set of Movie Toolbox functions, the QT atom functions, make it easy to create and manipulate data in this format. For more information on QT atoms and atom containers, see the chapter "Movie Toolbox" in this book.

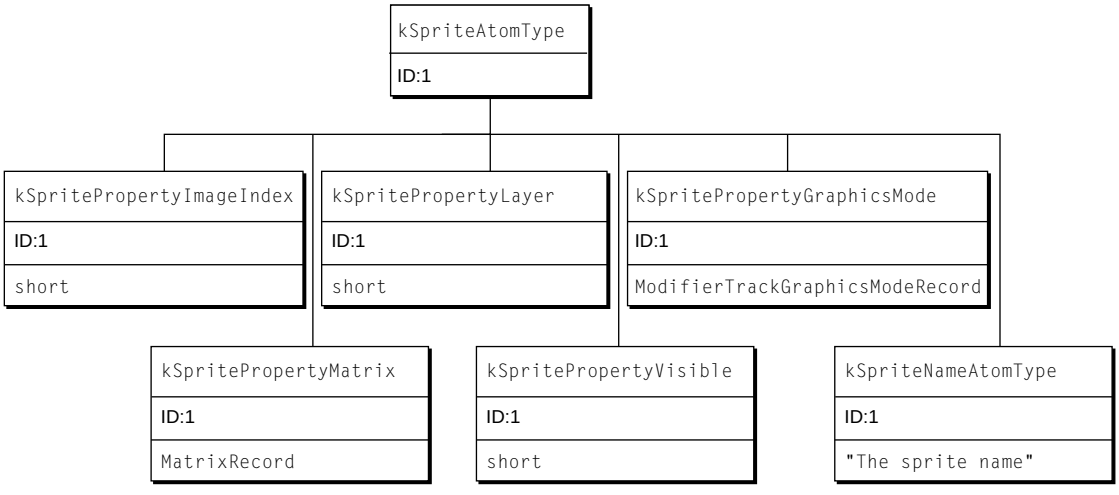
Figure 14-1 shows the high-level structure of a sprite track key frame sample. A key frame sample is represented by a QT atom container. Each atom in the atom container is represented by its atom type, atom ID, and, if it is a leaf atom, the type of its data.

Figure 14-1 A key frame sample atom container



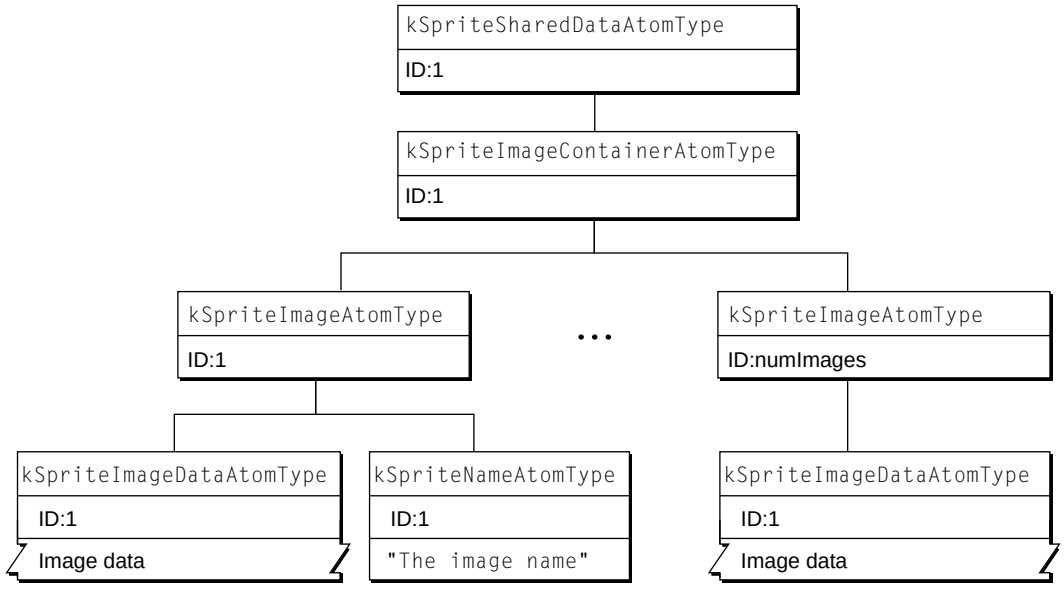
The QT atom container contains one child atom for each sprite in the key frame sample. Each sprite atom has a type of `kSpriteAtomType`. The sprite IDs are numbered from one to the number of sprites defined by the key frame sample (`numSprites`). Each sprite atom contains leaf atoms that define the properties of the sprite, as shown in Figure 14-2. For example, the `kSpritePropertyLayer` property defines a sprite's layer. Each sprite property atom has an atom type that corresponds to the property and an ID of 1.

Figure 14-2 Atoms that describe a sprite and its properties



In addition to the sprite atoms, the QT atom container contains one atom of type `kSpriteSharedDataAtomType` with an ID of 1. The atoms contained by the shared data atom describe data that is shared by all sprites. The shared data atom contains one atom of type `kSpriteImagesContainerAtomType` with an ID of 1 (Figure 14-3). The image container atom contains one atom of type `kImageAtomType` for each image in the key frame sample. The image atom IDs are numbered from one to the number of images (`numImages`). Each image atom contains a leaf atom that holds the image data (type `kSpriteImageDataAtomType`) and an optional leaf atom (type `kSpriteNameAtomType`) that holds the name of the image.

Figure 14-3 Atoms that describe sprite images

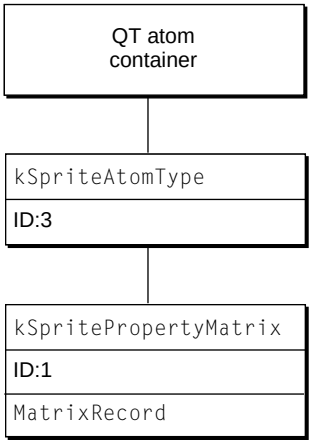


The format of an override sample is identical to that of a key frame sample with the following exceptions.

- An override sample does not contain images, which means it does not contain an atom of type `kSpriteImagesContainerAtomType` or any of its children.
- In an override sample, all of the sprite atoms and sprite property atoms are optional.

For example, to define an override sample that modifies the location of the third sprite defined by the previous key frame sample, you would create a QT atom container and add the following atoms to it (assuming that the sprite track format is of type `kKeyFrameAndSingleOverride`):

Figure 14-4 An example of an override sample atom container



Sprite Track Properties

In addition to defining properties for individual sprites, you can also define properties that apply to an entire sprite track. These properties may override default behavior or provide hints to the sprite media handler. The following sprite track properties are supported:

- The `kSpriteTrackPropertyBackgroundColor` property specifies a background color for the sprite track. The background color is used for any area that is not covered by regular sprites or background sprites. If you do not specify a background color, the sprite track uses black as the default background color.
- The `kSpriteTrackPropertyOffscreenBitDepth` property specifies a preferred bit depth for the sprite track's offscreen buffer. The allowable values are 8 and 16. To save memory, you should set the value of this property to the minimum depth needed. If you do not specify a bit depth, the sprite track allocates an offscreen buffer with the depth of the deepest intersecting monitor.
- The `kSpriteTrackPropertySampleFormat` property specifies the sample format for the sprite track. If you do not specify a sample format, the sprite track uses the default format, `kKeyFrameAndSingleOverride`.

To specify sprite track properties, you create a single QT atom container and add a leaf atom for each property you want to specify. To add the properties to

a sprite track, you call the new media handler function `SetMediaPropertyAtom`. To retrieve a sprite track's properties, you call the media handler function `GetMediaPropertyAtom`.

The sprite track properties and their corresponding atom data are outlined in Table 14-1.

Table 14-1 Sprite track properties

Atom type	Atom ID	Leaf data type
`kSpriteTrackPropertyBackgroundColor`	1	`RGBColor`
`kSpriteTrackPropertyOffscreenBitDepth`	1	`unsigned short`
`kSpriteTrackPropertySampleFormat`	1	`long`

Alternate Sources for Sprite Image Data

A sprite in a sprite track can obtain its image data from sources other than the images in the sprite track's key frame sample. The alternate image data overrides a particular image index in the sprite track so that all sprites with that image index will use the image data provided by the alternate source.

A sprite track can receive image data from another track within the same movie, called a modifier track. This is useful for compositing traditional video tracks with sprites. For example, you might create a sprite track in which sprite characters are watching television. The sprite track can receive video from another track, called a modifier track, to use as the image data for the television screen sprite. Other sprites can move in front of and behind the television. A sprite track can have more than one modifier track feeding it image data and more than one sprite can use the image data from a modifier track at one time.

In order for a sprite to receive image data from a modifier track, you must call the `AddTrackReference` function to link the modifier track to the sprite track that it modifies. In addition, you must update the sprite media's input map with an atom that specifies the input type (`kTrackModifierTypeSpriteImage`) and an atom that specifies the index of the image to replace (`kSpritePropertyImageIndex`).

A sprite track can also receive sprite image data from an application. For example, an application might provide live, digitized video data to a sprite track by calling `MediaSetNonPrimarySourceData`.

In addition to receiving image data, a sprite track can receive modifier track data to control its sprites. Currently, three kinds of modifier inputs are supported: images from a video track (`kTrackModifierTypeImage`), a matrix from a base track (`kTrackModifierObjectMatrix`), and a graphics mode from a base track (`kTrackModifierObjectGraphicsMode`).

For example, a modifier track can send matrices to individual sprites to control their locations. To do this, set up a modifier track, such as a tween track, to send matrix data to the sprite track. You must update the sprite media's input map with an atom that specifies the input type (`kTrackModifierObjectMatrix`) and an atom that specifies the ID of the sprite to replace (`kTrackModifierObjectID`). If the sprite track also contains matrices to move the sprites, the results are undefined.

For background information on modifier tracks, see the chapter "Movie Toobox" in this book.

Using the Sprite Media Handler

The sprite media handler provides functions that allow an application to create and manipulate a sprite animation as a track in a QuickTime movie.

The following sections are illustrated with code from the sample program `MakeSpriteMovie.c`, which shows how to create a QuickTime sprite track. The sample code creates a 640 by 480 pixel movie with one sprite track. The sprite track can have either a static background picture sprite or a solid color background. The sprite track has three other sprites whose properties change over time. The sprite track's media contains one key frame sample followed by many override samples. The key frame sample contains all of the images used by the sprites. The override samples contain the overrides of the locations, image indices, and layers needed to modify the sprites.

- "Defining a Key Frame Sample" discusses the tasks you must perform to create a key frame sample and add sprites and sprite image data to it
- "Defining Override Samples" describes how to create override samples and add them to the sprite track

- "Setting Properties of the Sprite Track" discusses how to set global characteristics of the sprite track
- "Getting Sprite Data From a Modifier Track" describes how to replace a sprite's image data with a modifier track

Defining a Key Frame Sample

In order to create a sprite track in a QuickTime movie, you must first create the movie ifself, a track to contain the sprites, and the track's media. Then, you define a key frame sample. A key frame sample defines the number of sprites, their initial property values, and the shared image data used by the sprites in the key frame sample and in all override samples that follow the key frame sample. The sample code discussed in this section creates a single key frame sample; however, a sprite track may contain multiple key frame samples, each with its own override samples.

Creating the Movie, Sprite Track, and Media

Listing 14-1 shows a code fragment from the sample code function `CreateSampleSpriteMovie`. This function creates a new movie file and calls another sample code function, `AddSpriteTrackToMovie`, which is responsible for creating a sprite track and adding it to the movie.

Listing 14-1 Creating a sprite track movie

```
// global constants
#define kSpriteTrackWidth 640
#define kSpriteTrackHeight 480


Movie       theMovie = nil;

// ...
// create a movie file
// ...

// add the sprite track to the movie
FailOSErr (AddSpriteTrackToMovie (theMovie, kSpriteTrackWidth,
    kSpriteTrackHeight, true));
```

The following code fragment from `AddSpriteTrackToMovie` (Listing 14-2) creates a new track and media, and calls `BeginMediaEdits` to prepare to add samples to the track's media.

Listing 14-2 Creating a track and media

```
// global constants
#define kSpriteMediaTimeScale 600


newTrack = NewMovieTrack (theMovie, ((long)trackWidth << 16),
    ((long)trackHeight << 16), 0);
newMedia = NewTrackMedia (newTrack, SpriteMediaType,
    kSpriteMediaTimeScale, nil, 0);

FailOSErr (BeginMediaEdits (newMedia));
```

Adding Images to the Key Frame Sample

Listing 14-3 shows the first part of the `AddSpriteTrackToMovie` function. The first task the function performs is to create a QT atom container to hold the key frame sample and add all of the images to be used for sprites to the key frame sample. For each image, this function calls the sample code function `AddPICTImageToKeyFrameSample` to compress the PICT image and then adds the compressed image to the key frame sample. The last parameter passed to the `AddPICTImageToKeyFrameSample` is the index of the image.

Listing 14-3 Adding images to the key frame sample

```
// global constants
#define kIconPictID 127
#define kWorldPictID 128
#define kBackgroundPictID 158
#define kFirstSpaceShipPictID (kBackgroundPictID + 1)
#define kNumSpaceShipImages 24


RGBColor keyColor;
```

```
keyColor.red = keyColor.green = keyColor.blue = 0xFFFF;// white

// create an empty key frame sample
FailOSErr (QTNewAtomContainer(&sample));

// add images to the key frame sample
err = AddPICTImageToKeyFrameSample (sample, kIconPictID,
    &keyColor, 1);
err = AddPICTImageToKeyFrameSample (sample, kWorldPictID,
    &keyColor, 2);
err = AddPICTImageToKeyFrameSample (sample, kBackgroundPictID,
    &keyColor, 3);
for ( i = 1; i <= kNumSpaceShipImages; i++ )
    err = AddPICTImageToKeyFrameSample (sample,
        kFirstSpaceShipPictID + i - 1, &keyColor, i + 3);
```

The `AddPICTImageToKeyFrameSample` function (Listing 14-4) calls another sample code function, `MakePictTransparent`, which strips any surrounding background color from a PICT image. `MakePictTransparent` does this by using the animation compressor to recompress the PICT image using a key color.

`AddPICTImageToKeyFrameSample` then calls the sample code function `ExtractCompressData`, which extracts the compressed image data from the PICT image. Finally, `AddPICTImageToKeyFrameSample` calls the sample code function `AddCompressedImageToKeyFrameSample`, which is responsible for preparing the compressed image data and adding it to the key frame sample.

Listing 14-4 The `AddPICTImageToKeyFrameSample` function

```
OSErr AddPICTImageToKeyFrameSample (QTAtomContainer keySample,
    short pictID, RGBColor *keyColor, short id)
{
    OSErr                   err = noErr;
    PicHandle               picture;
    Handle                  compressedPicture;
    ImageDescriptionHandle  idh;

    // get picture from resource
    picture = (PicHandle) GetPicture (pictID);
```

```
    DetachResource ((Handle)picture);

    // make the PICT "transparent"
    MakePictTransparent (picture, keyColor);
    // extract the compressed image data from the PICT
    ExtractCompressData (picture, &compressedPicture, &idh);

    // prepare the compressed image and add it to the key frame sample
    HLock (compressedPicture);
    AddCompressedImageToKeyFrameSample (keySample, idh,
        GetHandleSize (compressedPicture), *compressedPicture, id);

bail:
    if (picture)
        KillPicture (picture);
    if (compressedPicture)
        DisposeHandle( compressedPicture );
    if (idh)
        DisposeHandle ((Handle)idh);
    return err;
}
```

The `AddCompressedImageToKeyFrameSample` function (Listing 14-5) converts the compressed image to the appropriate format to be used by a sprite by appending the compressed image to an image description handle. Then, the function adds the appropriate atoms to represent the sprite image to the key frame sample:

- The function ensures that the sample contains a sprite shared data atom (atom type `kSpriteSharedDataAtomType`) with a child image container atom (atom type `kSpriteImagesContainerAtomType`).
- The function inserts a sprite image atom (atom type `kSpriteImageAtomType`) as a child of the image container atom.
- The function inserts a leaf atom as a child of the sprite image atom. The leaf atom's data is the concatenated image description handle and image.

Listing 14-5 The `AddCompressedImageToKeyFrameSample` function

```
OSErr AddCompressedImageToKeyFrameSample (QTAtomContainer keySample,
    ImageDescriptionHandle idh, long dataSize, Ptr compressedDataPtr,
    QTAtomID imageID)
{
    OSErr   err = noErr;
    Handle  imageData;
    QTAtom  defaultsAtom, imagesContainerAtom, imageAtom;

    // append compressed picture data to imageDescription to
    // obtain sprite image data
    FailMemErr (imageData = NewHandle(0));
    FailMemErr (HandAndHand((Handle)idh, imageData));
    FailMemErr (PtrAndHand (compressedDataPtr, imageData, dataSize));

    // if no kSpriteSharedDataAtomType atom in key sample, add one
    if ((defaultsAtom = QTFindChildByIndex (keySample, 0,
        kSpriteSharedDataAtomType, 1, nil)) == 0)
        FailOSErr (QTInsertChild (keySample, 0,
            kSpriteSharedDataAtomType, 1, 0, 0, nil, &defaultsAtom));

    // if no kSpriteImagesContainerAtomType in key sample, add one
    if ((imagesContainerAtom = QTFindChildByIndex (keySample,
        defaultsAtom, kSpriteImagesContainerAtomType, 1, nil)) == 0)
        FailOSErr (QTInsertChild (keySample, defaultsAtom,
            kSpriteImagesContainerAtomType, 1, 0, 0, nil,
            &imagesContainerAtom));

    // add the image and image data atoms to the key sample
    FailOSErr (QTInsertChild(keySample, imagesContainerAtom,
        kSpriteImageAtomType, imageID, 0, 0, nil, &imageAtom));
    HLock (imageData);
    FailOSErr (QTInsertChild (keySample, imageAtom,
        kSpriteImageDataAtomType, 1, 0, GetHandleSize(imageData),
        *imageData, nil));

bail:
    if (imageData)
```

```
        DisposeHandle (imageData);
    return err;
}
```

Adding Sprites to the Key Frame Sample

The `AddSpriteTrackToMovie` function adds the sprites with their initial property values to the key frame sample, as shown in Listing 14-6. If the `withBackgroundPicture` parameter is true, the function adds a background sprite. The function initializes the background sprite's properties, including setting the layer property to `kBackgroundSpriteLayerNum` to indicate that the sprite is a background sprite. The function calls `SetSpriteData` (Listing 14-7), which adds the appropriate property atoms to the `spriteData` atom container. Then, `AddSpriteTrackToMovie` calls `AddSpriteToSample` (Listing 14-8) to add the atoms in the `spriteData` atom container to the key frame sample atom container.

`AddSpriteTrackToMovie` adds the other sprites to the key frame sample and then calls `AddSpriteSampleToMedia` (Listing 14-9) to add the key frame sample to the media.

Listing 14-6 Adding sprites to the key frame

```
// global constants
#define kBackgroundImageIndex 3
#define kFirstSpaceShipImageIndex 4
#define kSpriteMediaFrameDuration 8


FailOSErr (QTNewAtomContainer (&spriteData));

if (withBackgroundPicture)
{
    // add background sprite
    location.h = 0;
    location.v = 0;
    visible = true;
    layer = kBackgroundSpriteLayerNum;
    imageIndex = kBackgroundImageIndex;
    SetSpriteData (spriteData, &location, &visible, &layer, &imageIndex);
    err = AddSpriteToSample (sample, spriteData, 1);
```

```
}

// add space ship sprite
location.h = 0;
location.v = 60;
visible = true;
layer = -1;
imageIndex = kFirstSpaceShipImageIndex;
SetSpriteData (spriteData, &location, &visible, &layer, &imageIndex);
err = AddSpriteToSample (sample, spriteData, 2);

// ...
// add other sprites
// ...

err = AddSpriteSampleToMedia (newMedia, sample,
    kSpriteMediaFrameDuration, true);
```

For each new property value that is passed into it as a parameter, the `SetSpriteData` function (Listing 14-7) calls `QTFindChildByIndex` to find the appropriate property atom. If the property atom already exists in the QT atom container, `SetSpriteData` calls `QTSetAtomData` to update the property's value. If the property atom does not exist in the container, `SetSpriteData` calls `QTInsertChild` to insert a new property atom.

Listing 14-7 The `SetSpriteData` function

```
OSErr SetSpriteData (QTAtomContainer sprite, Point *location,
    short *visible, short *layer, short *imageIndex)
{
    OSErr   err = noErr;
    QTAtom  propertyAtom;

    if (location) {
        MatrixRecordmatrix;

        // set up the value for the matrix property
        SetIdentityMatrix (&matrix);
        matrix.matrix[2][0] = ((long)location->h << 16);
        matrix.matrix[2][1] = ((long)location->v << 16);
```

```
        // if no matrix atom is in the container, insert a new one
        if ((propertyAtom = QTFindChildByIndex (sprite, 0,
            kSpritePropertyMatrix, 1, nil)) == 0)
            FailOSErr (QTInsertChild (sprite, 0, kSpritePropertyMatrix,
                1, 0, sizeof(MatrixRecord), &matrix, nil))
        // otherwise, replace the atom's data
        else
            FailOSErr (QTSetAtomData (sprite, propertyAtom,
                sizeof(MatrixRecord), &matrix));
    }

    // ...
    // handle other properties in a similar fashion
    // ...

    return err;
}
```

The `AddSpriteToSample` function (Listing 14-8) checks to see whether a sprite has already been added to a sample. If not, the function calls `QTInsertChild` to create a new sprite atom in the atom container that represents the sample. Then, `AddSpriteToSample` calls `QTInsertChildren` to insert the atoms in the sprite atom container as children of the newly created atom in the sample container.

Listing 14-8 The `AddSpriteToSample` function

```
OSErr AddSpriteToSample (QTAtomContainer theSample,
    QTAtomContainer theSprite, short spriteID)
{
    OSErr err = noErr;
    QTAtom newSpriteAtom;

    FailIf (QTFindChildByID (theSample, 0, kSpriteAtomType, spriteID,
        nil), paramErr);

    FailOSErr (QTInsertChild (theSample, 0, kSpriteAtomType, spriteID,
        0, 0, nil, &newSpriteAtom)); // index of zero means append
    FailOSErr (QTInsertChildren (theSample, newSpriteAtom, theSprite));
```

```
bail:
    return err;
}
```

The `AddSpriteSampleToMedia` function, shown in Listing 14-9, calls `AddMediaSample` to add either a key frame sample or an override sample to the sprite media.

Listing 14-9 The `AddSpriteSampleToMedia` function

```
OSErr AddSpriteSampleToMedia (Media theMedia, QTAtomContainer sample,
    TimeValue duration, Boolean isKeyFrame)
{
    OSErr err = noErr;
    SampleDescriptionHandle sampleDesc = nil;

    FailMemErr (sampleDesc = (SampleDescriptionHandle) NewHandleClear(
        sizeof(SampleDescription)));

    FailOSErr (AddMediaSample (theMedia, (Handle) sample, 0,
        GetHandleSize(sample), duration, sampleDesc, 1,
        isKeyFrame ? 0 : mediaSampleNotSync, nil));

bail:
    if (sampleDesc)
        DisposeHandle ((Handle)sampleDesc);

    return err;
}
```

Defining Override Samples

Once you have defined a key frame sample for the sprite track, you can add any number of override samples to modify sprite properties.

Listing 14-10 shows the portion of the `AddSpriteTrackToMovie` function that adds override samples to the sprite track to make the sprites appear to spin and move. For each override sample, the function modifies the space ship sprite's image index and location. The function calls `SetSpriteData` to update

the appropriate property atoms in the sprite atom container. Then, the function calls `AddSpriteToSample` to add the sprite atom container to the sample atom container. After all of the modifications have been made to the override sample, the function calls `AddSpriteSampleToMedia` to add the override sample to the media.

After adding all of the override samples to the media, `AddSpriteTrackToMovie` calls `EndMediaEdits` to indicate that it is done adding samples to the media. Then, `AddSpriteTrackToMovie` calls `InsertMediaIntoTrack` to insert the new media segment into the track.

Listing 14-10 Adding override samples

```
// global constants
#define kNumOverrideSamples 199
#define kFirstSpaceShipImageIndex 4
#define kNumSpaceShipImages 24
#define kLastSpaceShipImageIndex (kFirstSpaceShipImageIndex +
    kNumSpaceShipImages - 1)
#define kSpriteMediaFrameDuration 8


imageIndex = kFirstSpaceShipImageIndex;
location.h = 0;
location.v = 80;
// for each override sample
for ( i = 1; i < kNumOverrideSamples; i++ )
{
    // clear out the sample and spriteData atom containers
    QTRemoveChildren (sample, 0);
    QTRemoveChildren (spriteData, 0);

    // bump the space ship's image index every third frame to spin
    if ( (i % 3) == 0 ) {

        imageIndex++;
        if (imageIndex > kLastSpaceShipImageIndex)
            imageIndex = kFirstSpaceShipImageIndex;
    }
```

```
    // bump location of space ship sprite in each frame by one pixel
    // vertically and two horizontally
    location.h += 2;
    location.v++;

    // add the space ship sprite to the override sample
    SetSpriteData (spriteData, &location, nil, nil, &imageIndex);
    err = AddSpriteToSample (sample, spriteData, 2);

    // ...
    // make modifications to other sprites and add to the override sample
    // ...

    // add the override sample to the media
    err = AddSpriteSampleToMedia (newMedia, sample,
        kSpriteMediaFrameDuration, false);
}

EndMediaEdits (newMedia);
InsertMediaIntoTrack (newTrack, 0, 0, GetMediaDuration (newMedia),
    0x010000);
```

Setting Properties of the Sprite Track

In addition to adding key frame samples and override samples to the sprite track, you may want to set one or more global properties of the sprite track. For example, if you want to define a background color for your sprite track, you must set the sprite track's background color property. You do this by creating a leaf atom of type `kSpriteTrackPropertyBackgroundColor` whose data is the desired background color.

After adding the override samples, `AddSpriteTrackToMovie` adds a background color to the sprite track, as shown in Listing 14-11. If the `withBackgroundPicture` parameter is false, this function defines a solid background color for the sprite track. The function calls `QTNewAtomContainer` to create a new atom container for sprite track properties. `AddSpriteTrackToMovie` adds a new atom of type `kSpriteTrackPropertyBackgroundColor` to the container and calls `SetMediaPropertyAtom` to set the sprite track's properties.

Listing 14-11 Defining a background color

```
// add a background color to the sprite track
if (withBackgroundPicture == false)
{
    QTAtomContainer trackProperties;
    RGBColor backgroundColor;

    backgroundColor.red = 0x8000;
    backgroundColor.green = 0;
    backgroundColor.blue = 0xffff;

    // create a new atom container for sprite track properties
    QTNewAtomContainer (&trackProperties);

    // add an atom for the background color property
    QTInsertChild (trackProperties, 0,
        kSpriteTrackPropertyBackgroundColor, 1, 1, sizeof(RGBColor),
        &backgroundColor, nil);

    // set the sprite track's properties
    err = SetMediaPropertyAtom (newMedia, trackProperties);

    QTDisposeAtomContainer(trackProperties);
}
```

Getting Sprite Data From a Modifier Track

The sample program `AddReferenceTrack.c` illustrates how you can modify a movie to use a modifier track for a sprite's image data. The sample program prompts the user for a movie that contains a single sprite track. Then, it adds a track from a second movie to the original movie as a modifier track. The modifier track overrides the image data for a selected image index.

Listing 14-12 shows the first part of the main function of the sample program. It performs the following tasks:

- It loads the movie containing the sprite track.
- It calls `GetMovieTrackCount` to determine the total number of tracks in the sprite track movie.

■ It loads the movie containing the modifier track (`movieB`).

Listing 14-12 Loading the movies

```
OSErr               err;
short               movieResID = 0, resFref, resID = 0, resRefNum;
StandardFileReply   reply;
SFTypeList          types;
Movie               m;
FSSpec              fss;
Movie               movieB;
long                origTrackCount;

// prompt for a movie containing a sprite track and load it
types[0] = MovieFileType;
StandardGetFilePreview (nil, 1, types, &reply);
if (!reply.sfGood) return;

err = OpenMovieFile (&reply.sfFile, &resFref, fsRdPerm);
if (err) return;

err = NewMovieFromFile (&m, resFref, &movieResID, (StringPtr)nil,
    newMovieActive, ni);
if (err) return;

CloseMovieFile (resFref);

// get the number of tracks
origTrackCount = GetMovieTrackCount (m);

// load the movie to be used as a modifier track
FSMakeFSSpec (reply.sfFile.vRefNum, reply.sfFile.parID, "\pAdd Me",
    &fss);

err = OpenMovieFile (&fss, &resFref, fsRdPerm);
if (err) return;

err = NewMovieFromFile (&movieB, resFref, &resID, (StringPtr)nil, 0,
    nil);
```

```
if (err) return;

CloseMovieFile (resFref);
```

Once the two movies have been loaded, the sample program retrieves the first track, which is the sprite track, from the original movie, and sets the selection to the start of the movie (Listing 14-13). The sample program iterates through all the tracks in the modifier movie, disposing of all non-video tracks.

Next, the sample program calls `AddMovieSelection` to add the modifier track to the original movie. Finally, the sample program calls `AddTrackReference` to associate the modifier track with the sprite track it will modify. `AddTrackReference` returns an index of the added reference in the `referenceIndex` variable.

Listing 14-13 Adding the modifier track to the movie

```
Movie           m;
TimeValue       oldDuration;
Movie           movieB;
long            i, origTrackCount, referenceIndex;
Track           newTrack, spriteTrack;

// get the first track in original movie and position at the start
spriteTrack = GetMovieIndTrack (m, 1);
SetMovieSelection (m, 0 ,0);

// remove all tracks except video in modifier movie
for (i = 1; i <= GetMovieTrackCount (movieB); i++)
{
    Track t = GetMovieIndTrack (movieB, i);
    OSType aType;

    GetMediaHandlerDescription (GetTrackMedia(t), &aType, nil, nil);
    if (aType != VideoMediaType)
    {
        DisposeMovieTrack (t);
        i--;
    }
}
```

```
// add the modifier track to original movie
oldDuration = GetMovieDuration (m);
AddMovieSelection (m, movieB);
DisposeMovie (movieB);

// truncate the movie to the length of the original track
DeleteMovieSegment (m, oldDuration,
    GetMovieDuration (m) - oldDuration);

// associate the modifier track with the original sprite track
newTrack = GetMovieIndTrack (m, origTrackCount + 1);
AddTrackReference (spriteTrack, newTrack, kTrackModifierReference,
    &referenceIndex);
```

In addition to adding a reference to the modifier track, the sample program must update the sprite media's input map to describe how the modifier track should be interpreted by the sprite track. The sample program performs the following tasks (Listing 14-14):

- It retrieves the sprite track's media by calling `GetTrackMedia`
- It calls `GetMediaInputMap` to retrieve the media's input map.
- It adds a parent atom to the input map of type `kTrackModifier` input. The ID of the atom is the reference index retrieved by the `AddTrackReference` function.
- It adds two child atoms, one that specifies that the input type of the modifier track is of type `kTrackModifierTypeSpriteImage`, and one that specifies the index of the sprite image to override.
- It calls `SetMediaInputMap` to update the media's input map.

Listing 14-14 Updating the media's input map

```
#define kImageIndexToOverride 1

Movie           m, movieB;
long            referenceIndex, imageIndexToOverride;
Track           spriteTrack;
QTAtomContainerinputMap;
```

```
QTAtom          inputAtom;
OSType          inputType;
Media           spriteMedia;

// get the sprite media's input map
spriteMedia = GetTrackMedia (spriteTrack);
GetMediaInputMap (spriteMedia, &inputMap);

// add an atom for a modifier track
QTInsertChild (inputMap, kParentAtomIsContainer,
    kTrackModifierInput, referenceIndex, 0, 0, nil, &inputAtom);

// add a child atom to specify the input type
inputType = kTrackModifierTypeSpriteImage;
QTInsertChild (inputMap, inputAtom, kTrackModifierType, 1, 0,
    sizeof(inputType), &inputType, nil);

// add a second child atom to specify index of image to override
imageIndexToOverride = kImageIndexToOverride;
QTInsertChild (inputMap, inputAtom, kSpritePropertyImageIndex, 1, 0,
    sizeof(imageIndexToOverride), &imageIndexToOverride, nil);

// update the sprite media's input map
SetMediaInputMap (spriteMedia, inputMap);
QTDisposeAtomContainer (inputMap);
```

Once the media's input map has been updated, the application can save the movie.

Sprite Media Handler Reference

Constants

Sprite Track Formats

The following constants represent formats of a sprite track. The value of the constant indicates how override samples in a sprite track should be interpreted. You set a sprite track's format by creating a `kSpriteTrackPropertySampleFormat` atom.

```
enum {
    kKeyFrameAndSingleOverride                  = 1L << 1,
    kKeyFrameAndAllOverrides                    = 1L << 2
};
```

Constant descriptions

`kKeyFrameAndSingleOverride`
: The current state of the sprite track is defined by the most recent key frame sample and the current override sample. This is the default format.

`kKeyFrameAndAllOverrides`
: The current state of the sprite track is defined by the most recent key frame sample and all subsequent override samples up to and including the current override sample.

Sprite Media Atom Types

The following string constants represent atom types for sprite media.

```
enum {
    kSpriteAtomType                     = 'sprt',
    kSpriteImagesContainerAtomType      = 'imct',
    kSpriteImageAtomType                = 'imag',
```

```
    kSpriteImageDataAtomType            = 'imda',
    kSpriteSharedDataAtomType           = 'dflt',
    kSpriteNameAtomType                 = 'name'
    kSpritePropertyMatrix               = 1
    kSpritePropertyVisible              = 4
    kSpritePropertyLayer                = 5
    kSpritePropertyGraphicsMode         = 6
    kSpritePropertyImageIndex           = 101
    kSpritePropertyBackgroundColor      = 101
    kSpritePropertyOffscreenBitDepth    = 102
    kSpritePropertySampleFormat         = 103
};
```

Constant descriptions

`kSpriteAtomType` The atom is a parent atom that describes a sprite. It contains atoms that describe properties of the sprite. Optionally, it may also include an atom of type `kSpriteNameAtomType` that defines the name of the sprite.

`kSpriteImagesContainerAtomType`
The atom is a parent atom that contains atoms of type `kSpriteImageAtomType`.

`kSpriteImageAtomType`
The atom is a parent atom that contains an atom of type `kSpriteImageDataAtomType`. Optionally, it may also include an atom of type `kSpriteNameAtomType` that defines the name of the image.

`kSpriteImageDataAtomType`
The atom is a leaf atom that contains image data.

`kSpriteSharedDataAtomType`
The atom is a parent atom that contains shared sprite data, such as an atom container of type `kSpriteImagesContainerAtomType`.

`kSpriteNameAtomType` The atom is a leaf atom that contains the name of a sprite or an image. The leaf data is composed of one or more ASCII characters.

`kSpritePropertyImageIndex`
A leaf atom containing the image index property which is of type `short`. This atom is a child atom of the `kSpriteAtom`.

`kSpritePropertyLayer`

A leaf atom containing the layer property which is of type `short`. This atom is a child atom of the `kSpriteAtom`.

`kSpritePropertyMatrix`

A leaf atom containing the matrix property which is of type `MatrixRecord`. This atom is a child atom of the `kSpriteAtom`.

`kSpritePropertyVisible`

A leaf atom containing the visible property which is of type `short`. This atom is a child atom of the `kSpriteAtom`.

`kSpritePropertyGraphicsMode`

A leaf atom containing the matrix property which is of type `ModifyerTrackGraphicsModeRecord`. This atom is a child atom of the `kSpriteAtom`.

`kSpritePropertyBackgroundColor`

A leaf atom containing the background color property which is of type `RGBColor`. This atom is used in a sprite track's `MediaPropertyAtom` atom container.

`kSpritePropertyOffscreenBitDepth`

A leaf atom containing the preferred offscreen bitdepth which is of type `short`. This atom is used in a sprite track's `MediaPropertyAtom` atom container.

`kSpritePropertySampleFormat`

A leaf atom containing the sample format property which is of type `short`. This atom is used in a sprite track's `MediaPropertyAtom` atom container.

Sprite Media Handler Functions

SetSpriteMediaSpriteProperty

The `SetSpriteMediaSpriteProperty` function sets the specified property of a sprite.

```
pascal ComponentResult SetSpriteMediaSpriteProperty (
                    MediaHandler mh,
                    short spriteIndex,
                    long propertyType,
                    void* propertyValue);
```

`mh` Specifies the sprite media handler for this operation.

`spriteIndex` Specifies the index of the sprite to be modified.

`propertyType` Specifies the property to be set.

`propertyValue` Specifies the new value of the property.

DISCUSSION

You call this function to modify a property of a sprite. You set the `propertyType` parameter to the property you want to modify. You set the `spriteIndex` parameter to the index of the sprite whose property you want to set. The index must be between one and the number of available sprites. You can determine how many sprites are available by calling `CountSpriteMediaSprites`.

The type of data you pass for the `propertyValue` parameter depends on the property type. The following table lists the sprite properties and the data types of the corresponding property values.

Sprite Property	Data Type
`kSpritePropertyMatrix`	`MatrixRecord *`
`kSpritePropertyVisible`	`short`

Sprite Property	Data Type
kSpritePropertyLayer	short
kSpritePropertyGraphicsMode	ModifierTrackGraphicsModeRecord *
kSpritePropertyImageIndex	short

RESULT CODES

noErr	0	No error
invalidSpritePropertyErr	-2065	Specified sprite property does not exist
invalidSpriteIndexErr	-2067	Sprite index is out of range

GetSpriteMediaSpriteProperty

The `GetSpriteMediaSpriteProperty` function retrieves the value of the specified sprite property.

```
pascal ComponentResult GetSpriteMediaSpriteProperty (
                    MediaHandler mh,
                    short spriteIndex,
                    long propertyType,
                    void* propertyValue);
```

`mh` Specifies the sprite media handler for this operation.

`spriteIndex` Specifies the index of the sprite for this operation.

`propertyType` Specifies the property whose value should be retrieved.

`propertyValue`
On return, contains a pointer to the value of the property.

DISCUSSION

You call this function to retrieve a value of a sprite property. You set the `propertyType` parameter to the property you want to retrieve. You set the `spriteIndex` parameter to the index of the sprite whose property you want to retrieve. The index must be between one and the number of available sprites.

You can determine how many sprites are available by calling `CountSpriteMediaSprites`.

On return, the `propertyValue` parameter contains a pointer to the specified property's value; the data type of that value depends on the property. The following table lists the sprite properties and the data types of the corresponding property values.

Sprite Property	Data Type
`kSpritePropertyMatrix`	`MatrixRecord *`
`kSpritePropertyVisible`	`short *`
`kSpritePropertyLayer`	`short *`
`kSpritePropertyGraphicsMode`	`ModifierTrackGraphicsModeRecord *`
`kSpritePropertyImageIndex`	`short *`

RESULT CODES

`noErr`	0	No error
`invalidSpritePropertyErr`	-2065	Specified sprite property does not exist
`invalidSpriteIndexErr`	-2067	Sprite index is out of range

HitTestSpriteMedia

The `HitTestSpriteMedia` function determines whether any sprites are at a specified location.

```
pascal ComponentResult HitTestSpriteMedia (
                    MediaHandler mh,
                    long flags,
                    Point loc,
                    short* spriteHitIndex);
```

`mh`	Specifies the sprite media handler for this operation.
`flags`	Specifies flags to control the hit testing operation.
`loc`	Specifies a point in the coordinate system of the sprite track's movie to test for the existence of a sprite.

`spriteHitIndex`
Contains a pointer to a `short` integer. On return, this integer contains the index of the frontmost sprite at the location specified by `loc`. If no sprite exists at the location, the function sets the value of this parameter to 0.

DISCUSSION

You call this function to determine whether any sprites exist at a specified location in the coordinate system of a sprite track's movie. You can pass flags to this function to control the hit testing operation more precisely. For example, you may want the hit test operation to detect a sprite whose bounding box contains the specified location. The allowable flags for sprite hit testing are described in the chapter "Movie Toolbox" in this book.

CountSpriteMediaSprites

The `CountSpriteMediaSprites` function retrieves the number of sprites that currently exist in a sprite track.

```
pascal ComponentResult CountSpriteMediaSprites (
                    MediaHandler mh,
                    short* numSprites);
```

`mh` Specifies the sprite media handler for this operation.

`numSprites` Contains a pointer to a short integer. On return, this integer contains the number of sprites for the sprite media's current time.

DISCUSSION

This function determines the number of sprites that currently exist based on the key frame that is in effect.

CountSpriteMediaImages

The `CountSpriteMediaImages` function retrieves the number of images that currently exist in a sprite track.

```
pascal ComponentResult CountSpriteMediaImages (
                    MediaHandler mh,
                    short* numImages);
```

`mh` Specifies the sprite media handler for this operation.

`numImages` Contains a pointer to a short integer. On return, this integer contains the number of images for the sprite media's current time.

DISCUSSION

This function determines the number of images that currently exist based on the key frame that is in effect.

GetSpriteMediaIndImageDescription

The `GetSpriteMediaIndImageDescription` function retrieves an image description for the specified image in a sprite track.

```
pascal ComponentResult GetSpriteMediaIndImageDescription (
                    MediaHandler mh,
                    short imageIndex,
                    ImageDescriptionHandle imageDescription);
```

`mh` Specifies the sprite media handler for this operation.

`imageIndex` Specifies the index of the image whose image description should be retrieved.

`imageDescription`
Specifies an image description handle. On return, this handle contains the image description for the specified image.

DISCUSSION

You set the `imageIndex` parameter to the index of the image whose image description you want to retrieve. The index must be between one and the number of available images. You can determine how many images are available by calling `CountSpriteMediaImages`.

The handle specified by the `imageDescription` parameter must be unlocked; this function resizes the handle if necessary.

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified
`invalidImageIndexErr`	-2068	Image index is out of range

Memory Manager errors, as documented in *Inside Macintosh: Memory*.

GetDisplayedSampleNumber

The `GetDisplayedSampleNumber` function retrieves the number of the sample that is currently being displayed.

```
pascal ComponentResult GetDisplayedSampleNumber (
                    MediaHandler mh,
                    long* sampleNum);
```

`mh` Specifies the sprite media handler for this operation.

`sampleNum` Contains a pointer to a long integer. On return, this integer contains the number of the sample that is currently being displayed.

DISCUSSION

You call this function when you need to retrieve the sample number of the sample that is being displayed.

Preview Components

Contents

This chapter discusses new features and changes to preview components as documented in Chapter 12 of *Inside Macintosh: QuickTime Components*.

New Features of Preview Components

Single Fork Preview Support

QuickTime has always supported the display of file previews using the `StandardGetFilePreview` function. The format for storing these file previews has always been based on Macintosh resources. However, this approach does not work for files created or viewed on operating systems that do not support resource forks. Beginning with QuickTime 2.1, you can now use the `StandardGetFilePreview` function to display previews that are stored in the data fork of a file. QuickTime does not, however, provide support for creating previews that are stored in the data fork of a file. Applications must create these previews themselves.

Preview Components Reference

This section describes the new structure associated with preview components. This new structure makes it possible for your application to use `StandardGetFilePreview` to display previews stored entirely in the data fork of a file. Likewise, your application can create file previews and store them in the data fork of a file so they can be viewed by users of other operating systems.

Resources

The Preview Resource

If your application creates previews, you may want to write them using the data fork format so they can be used on any platform on which QuickTime is available.

The preview display code assumes that the data fork of the file is formatted using QuickTime atoms. See *QuickTime File Format Specification, May 1996* for information on atom-based storage.

Adding a preview results in at least two atoms being added to the data file. The first atom has a `pnot` tag. Its basic structure is the same as the `pnotResource` structure.

```
struct PreviewResourceRecord {
    unsigned long       modDate;
    short               version;
    OSType              resType;
    short               resID;
};
```

Field descriptions

`modDate` Contains the modification time (in the standard Macintosh format of seconds since midnight, January 1, 1904) of the file for which the preview was created. This parameter allows you to find out if the preview is out of date with the contents of the file.

`version` Contains the version number of the preview resource. The low bit of the version is a flag for preview components that only reference their data. If the bit is set, it indicates that the resource identified in the preview resource is not owned by the preview component, but is part of the file. It is not removed when the preview is updated or removed (using the Image Compression Manager's `MakeFilePreview` or `AddFilePreview` function), as it would if the version number were 0.

`resType` Identifies the type of the preview component used to display the preview data and the type of the atom containing the preview data.

`resID` Contains the index (1-based) of the atom to be used. For example, a `resType` of PICT and a `resID` of 2 tells QuickTime to use the second PICT atom in the file for the preview data.

Data Handler Components

Contents

This chapter describes data handler components. **Data handler components** allow QuickTime to retrieve time-based data from external storage devices and, in some cases, store time-based data on those devices.

This chapter is divided into the following sections:

- "About Data Handler Components" provides a general introduction to components of this type
- "Using Data Handler Components" describes how QuickTime uses these components
- "Creating a Data Handler Component" describes how to create one of these components
- "Reference to Data Handler Components" presents detailed information about the functions that are supported by these components

In most cases you do not need to create a data handler or use one directly, because QuickTime takes care of data storage and retrieval for you through its built-in media handlers. However, you may need to create a data handler component to read or write to a non-Macintosh storage medium.

Data handler components exist both in QuickTime for Macintosh and QuickTime for Windows. Much of the background information is common to both platforms. However, there are some important technical differences between data handler components for these two platforms, such as the technique you would use to create a component. Therefore, whenever appropriate, this chapter refers you to specific QuickTime for Windows documentation for additional information.

Data handler components rely on the facilities of the Component Manager. In order to create or use any component, your application must also use the Component Manager. If you are not familiar with the Component Manager, see Chapter 6 of *Inside Macintosh: More Macintosh Toolbox*. If you are developing for QuickTime for Windows, you should also be familiar with *Creating Custom Components: QuickTime for Windows*. In addition, you should be familiar with the Movie Toolbox documentation in this book and in *Inside Macintosh: QuickTime* .

Note
This chapter describes the interface provided in QuickTime and QuickTime for Windows versions 2.0 and later. In addition, unless noted otherwise, the data handler components supplied by Apple support the entire interface described in this note. ◆

About Data Handler Components

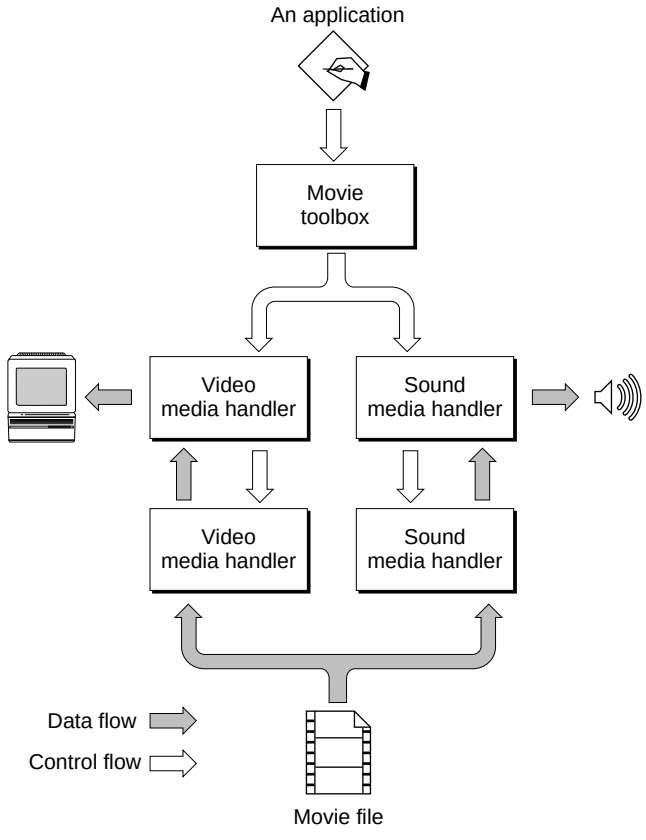
A data handler component stores and retrieves time-based data on a storage device, such as a movie file, on behalf of another QuickTime component, typically a media handler component or a sequence grabber component. Different QuickTime components are used depending on if you are retrieving or storing data.

Movie Playback

During data retrieval, such as playback of a movie, a media handler component isolates your application and the Movie Toolbox from the details of how to retrieve data from a particular storage medium. Therefore, unless you are writing your own media handler, you do not have to directly use data handler components in your application, the retrieval of your data will be taken care of for you by the media handler the Movie Toolbox calls. However, you can call the data handler directly if you need to explicitly tell the data handler something, such as to use less memory when caching QuickTime data. If you are reading from a non-Macintosh storage medium, or multiple storage media, you might need to write your own data handler.

Figure 16-1 shows the relationships between an application, the Movie Toolbox, QuickTime media handlers, and data handler components during movie playback. Notice that the media handlers intercept the data from the data handlers and plays it, while the Movie Toolbox controls the media handlers for your application.

Figure 16-1 Playing a movie



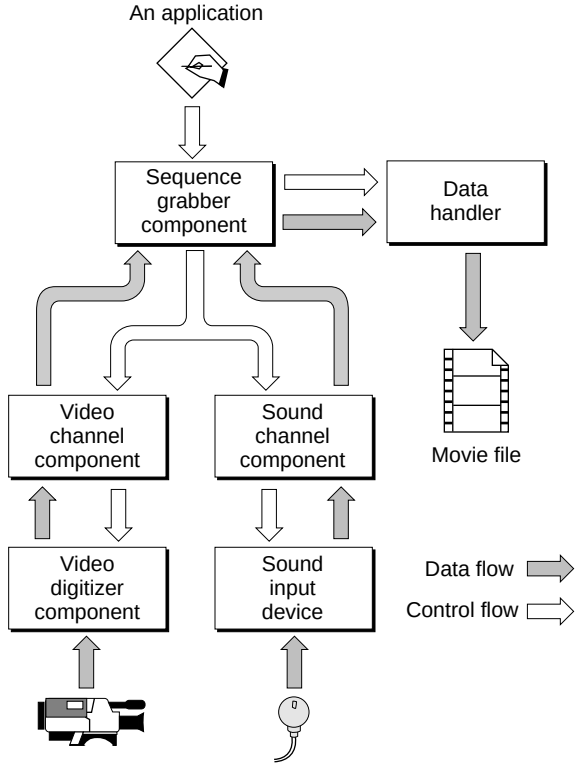
Movie Capture

During data storage, such as the capture of video and sound into a movie file, a a sequence grabber component isolates your application from the details of how to capture the raw data from a particular device. Therefore, during movie capture you do not have to directly use data handler components in your application, the storage of your data will be taken care of for you by the sequence grabber component you call. If, however, you are storing data onto a non-Macintosh or proprietary storage medium, or multiple storage media, you might need to write your own data handler.

The sequence grabber component calls the appropriate channel component, such as a video, sound, or text channel component, to retrieve the raw data from an input device, such as a microphone.

Figure 9-2 shows the relationships between an application, a sequence grabber component, two channel components, and a data handler while capturing movie data. Notice that the sequence grabber component intercepts the data from the channel components and then passes it on to a data handler, which writes it to a movie file. The sequence grabber controls the channel components and the data handler for your application.

Figure 16-2 Capturing movie data



Processing data

Data handlers do not know anything about the content of the data they process. It is the responsibility of the client (a media handler component or a channel component) to process the data. In the case of a movie's video data during movie playback, for example, the media handler takes the data from a data handler and uses the facilities of the Image Compression Manager to display the movie data on the computer screen. See *Inside Macintosh: QuickTime Components* for more information about media handlers.

While data handlers do not work with the content of the data they process, they must be aware of all of the details involved in storing and retrieving data from the storage medium they support. Apple provides several data handlers and a selection mechanism for choosing an appropriate handler. For example, one supports data access from HFS volumes and another supports the memory-based data handler, which allows QuickTime to retrieve movies from memory handles. These two data handler components use very different mechanisms to store and retrieve movie data.

You might need to write your own data handler when you are accessing a storage medium for which there is no Apple-supplied data handler or when playing movies from a multimedia server, as you will need to use a data handler that understands the network protocols and data formats necessary to communicate with that server.

Identifying Containers With Data References

A **container** is the system element that contains the movie data and can be any element that can contain data. For example, a container may be an in-memory data structure, a local disk file, or a file on a networked multimedia server. As is the case throughout QuickTime, all data handlers identify their movie-data containers with data references. Data references identify the location of the container and its type.

Different container types may require different types of references. For example, files are identified using aliases, while memory-based movies are identified by handles. The data reference data type is flexible enough to accommodate all these cases. The data handler component must specify the type of reference it requires and verify that the references supplied by client applications are valid. Data handler components use the component subtype value to specify the reference type they support.

Whenever an application opens a container, the Movie Toolbox determines the most appropriate data handler component to use in order to access that container. The Movie Toolbox makes this determination by querying the various data handlers installed on the user's computer. If your application uses the Movie Toolbox, this selection process is transparent to your program. If you develop your own data handler, your component must support the selection functions, see "Data Handler Components Reference" (page 16-28), for more information).

Using Data Handler Components

This section describes how applications use data handler components. You should read this section if you are writing your own media handler or your own data handler.

This section is divided into the following topics:

- "Selecting a Data Handler" describes the facilities that are available to help your application choose the best data handler for a given context.
- "Managing Data References" describes how your application gains access to a container using a data handler component.
- "Retrieving Movie Data" describes how your application reads movie data.
- "Storing Movie Data" describes how your application can write movie data using a data handler component.
- "Managing the Data Handler" describes your application's responsibilities while maintaining its connection with a data handler.

Selecting a Data Handler

To help developers choose the best data handler for a specific situation while still making it easy for an application to find a usable data handler, Apple has defined two separate and complementary mechanisms for selecting data handler components. You can use the Component Manager's selection mechanisms to find a data handler that meets your needs and you can interrogate a data handler to determine if it supports a specific data reference. Both mechanisms rely on characteristics of the current data reference in order to make the selection.

Before you can use a data handler component, your application must open a connection to that component. The easiest way to open a connection to a data handler component is to call the Movie Toolbox's `GetDataHandler` function. You supply a data reference and the Movie Toolbox selects an appropriate data handler component for you. This function is preferred for opening a data handler as it reliably chooses the best data handler. For more information about this function, see the chapter "Movie Toolbox" in *Inside Macintosh: QuickTime*.

Alternatively, you may use the Component Manager to open your connection. Call the Component Manager's `OpenDefaultComponent` or `OpenComponent` function to do so, but be aware that these functions are often unable to make the best choice when there are several different data handlers available for a file.

Selecting by Component Type Value

At the most basic level, your application can use the Component Manager's built-in selection mechanisms to find a data handler component for a data reference. You may use the Component Manager's `FindNextComponent` function in order to retrieve a list of all data handler components that meet your needs. You specify your request by supplying the component's characteristics in a component description record—in particular, in the `componentType`, `componentSubtype`, `componentManufacturer`, and `componentFlags` fields.

All data handler components have a component type value of `'dhlr'`, which is defined by the `dataHandlerType` constant. Data handler components use the value of the component subtype field to indicate the type of data reference they support. As a result of this convention, note that all data handlers that share a component subtype value must be able to recognize and work with data references of the same type. For example, file system data handlers always carry a component subtype value of `'alis'`, which indicates that their data references are file system aliases (note that this is true for QuickTime on the Macintosh and under Windows, even though there is not, properly, a file system alias under Windows). Apple's memory-based data handler for the Macintosh has a component subtype value of `'hndl'`.

Apple has not defined any special manufacturer field values or component flags values for data handler components. You may use the manufacturer field to select data handlers supplied by a specific vendor. To do so, you need to determine the appropriate manufacturer field value for that vendor.

Interrogating a Data Handler's Capabilities

While you can use the Component Manager's selection mechanisms to find a data handler component that can recognize data references of a specific type, your application must interact with the data handler in order to determine whether it can support a specific data reference. Apple has defined two functions, `DataHCanUseDataRef` and `DataHGetVolumeList`, that allow you to query a data handler component in order to find out whether it can work with a data reference. By using these two functions, your application can choose a data handler that is best-suited to its specific needs.

Using the `DataHCanUseDataRef` function, you supply a data reference to the data handler component. The component then reports what it can do with that data reference. The returned value indicates the level and, to some extent, the quality of service the data handler can provide (for example, whether the component can read data from or write data to the data reference and whether the component uses any special support when working with that data reference).

Because calling the `DataHCanUseDataRef` function in several data handlers can get time consuming, Apple has also defined a function that helps narrow the search. By using the `DataHGetVolumeList` function, your application can obtain a list of all the file system volumes that a data handler can support. In response to your request, the data handler returns the list and flags indicating the level and quality of service the data handler can provide for containers on that volume.

For more information on these functions, see "Selecting a Data Handler" (page 16-29).

Managing Data References

Once you have selected a data handler component, you must provide a data reference to the data handler. Use the `DataHSetDataRef` function to supply a data reference to a data handler. Once you have assigned a data reference to the data handler, your application may start reading and/or writing movie data from that data reference. The `DataHGetDataRef` function allows your application to obtain a data handler's current data reference.

Data handlers also provide a function that allows your application to determine whether two data references are equivalent (that is, refer to the same movie container). Your application provides a data reference to the `DataHCompareDataRef` function. The data handler returns a Boolean value

indicating whether that data reference matches the data handler's current data reference.

For more information on these functions, see "Working With Data References" (page 16-36).

Retrieving Movie Data

Before your application can read data using a data handler component, you must open a read path to the current data reference. Use the `DataHOpenForRead` function to request read access to the current data reference. Once you have gained read access to the data reference, data handlers provide both high- and low-level read functions.

The high-level function, `DataHGetData`, provides an easy-to-use, synchronous read interface. Being a synchronous function, `DataHGetData` does not return control to your application until the data handler has read and delivered the data you request.

If you need more control over the read operation, you can use the low-level function, `DataHScheduleData`, to issue asynchronous read requests. When you call this function, you provide detailed information specifying when you need the data from the request. The data handler returns control to your application immediately, and then processes the request when appropriate. When the data handler completes the request, it calls your data-handler completion function to report that the request has been satisfied, see "Completion Function" (page 16-61) for more information on the data-handler completion function.

Besides simply scheduling read requests that must be satisfied during a movie's playback, another use of the `DataHScheduleData` function is to prepare a movie for playback (commonly referred to as pre-rolling the movie). The `DataHScheduleData` function uses several special values to indicate a pre-roll operation. Your application calls the `DataHScheduleData` function one or more times to schedule the pre-roll read requests, and then uses the `DataHFinishData` function to tell the data handler to start delivering the requested data.

For more information on these functions and about pre-roll operations, see "Reading Movie Data" (page 16-41).

Storing Movie Data

Before your application can write data using a data handler component, you must open a write path to the current data reference. Use the `DataHOpenForWrite` function to request write access to the current data reference. Once you have gained write access to the data reference, data handler components provide both high- and low-level write functions.

Note
QuickTime for Windows version 2.1.1 or earlier does not support writing movie data. ◆

The high-level function, `DataHPutData`, allows you to easily append data to the end of the container identified by a data reference. Except when capturing movie data using the sequence grabber component, the Movie Toolbox uses this call when writing data to movie files. However, this function does not allow your application to write to any location other than the end of the container. In addition, this is a synchronous operation, so control is not returned to your program until the write is complete. As a result, this function is not well-suited to high-performance write operations, such as would be required to capture a movie.

If you need a more flexible write facility, or one with higher performance characteristics, you can use the `DataHWrite` function. This function is intended to support high-speed writes, suitable for movie capture operations. For example, Apple's sequence grabber component uses this data handler function to capture movies.

When you call this function, you provide detailed information specifying the location in the container that is to receive the data. The data handler returns control to your application immediately, and then processes the request asynchronously. When the data handler completes the request, it calls your data-handler completion function to report that the request has been satisfied, see "Completion Function" (page 16-61) for more information on the data-handler completion function.

In addition to the `DataHWrite` function, data handler components provide several other "helper" functions that allow you to create new movie containers and prepare them for a movie capture operation.

For more information on all of these functions, see "Writing Movie Data" (page 16-50).

Managing the Data Handler

Data handler components provide a number of functions that your application can use to manage its connection to the handler. The most important among these is `DataHTask`, which provides processor time to the handler. Your application should call this function often so that the handler has enough time to do its work.

Other functions in this category provide playback hints to the data handler and allow your application to influence how the component handles its cached data.

For more information on these functions, see "Managing Data Handler Components" (page 16-58).

Creating a Data Handler Component

This section describes the details of creating a data handler component and includes source code for a simple data handler component. After reading this section, you will understand all of the special requirements of these components. The functional interface that your component must support is described in "Data Handler Components Reference" (page 16-28).

You should consider developing your own data handler component if you are planning to provide a new type of movie container or a container that requires special data handling techniques. For example, if you are planning to develop a networked multimedia server, you would most likely need to develop a new data handler that could support the special protocols required by your server. By encapsulating that protocol support in a data handler, QuickTime applications can access the movie data on your server without having to implement any special support. In this way, your server becomes a seamless part of the user's system.

Before reading this section, you should be familiar with how to create components. See "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for a complete description of components, how to use them, and how to create them on the Macintosh. For further information about using the Component Manager with QuickTime for Windows, see *Creating Custom Components: QuickTime for Windows.*

General Information

All data handler components have a component type value of `'dhlr'`, which is defined by the `dataHandlerType` constant. Data handler components use the value of the component subtype field to indicate the type of data reference they support. As a result of this convention, note that all data handlers that share a component subtype value must be able to recognize and work with data references of the same type. For example, file system data handlers always carry a component subtype value of `'alis'`, which indicates that their data references are file system aliases (note that this is true for QuickTime on the Macintosh and under Windows, even though there is not, properly, a file system alias under Windows). Apple's memory-based data handler for the Macintosh has a component subtype value of `'hndl'`.

```
#define dataHandlerType 'dhlr'
#define rAliasType      'alis'
```

Apple has not defined any special manufacturer field values or component flags values for data handler components. Developers may use the manufacturer field value to select your data handler from among all the data handlers that support a given type of data reference.

Apple has defined a functional interface for data handler components. For information about the functions that your component must support, see "Data Handler Components Reference" (page 16-28). You can use the following constants to refer to the request codes for each of the functions that your component must support:

```
enum {

    kDataGetDataSelector    = 2,    /* DataHGetData */
    kDataPutDataSelector    = 3,    /* DataHPutData */
    kDataFlushDataSelector= 4,  /* DataHFlushData */
    kDataOpenForWriteSelector= 5,/* DataHOpenForWrite */
    kDataCloseForWriteSelector= 6,/* DataHCloseForWrite */
    kDataOpenForReadSelector= 8,/* DataHOpenForRead */
    kDataCloseForReadSelector= 9,/* DataHCloseForRead */
    kDataSetDatRefSelector= 10, /* DataHSetDataRef */
    kDataGetDataRefSelector= 11,/* DataHGetDataRef */
    kDataCompareDataRefSelector= 12,/* DataHCompareDataRef */
    kDataTaskSelector           = 13,   /* DataHTask */
    kDataScheduleDataSelector= 14,/* DataHScheduleData */
```

```
    kDataFinishDataSelector= 15,/* DataHFinishData */
    kDataFlushCacheSelector= 16,/* DataHFlushCache */
    kDataResolveDataRefSelector= 17,/* DataHResolveDataRef
*/
    kDataGetFileSizeSelector= 18,/* DataHGetFileSize */
    kDataCanUseDataRefSelector= 19,/* DataHCanUseDataRef */
    kDataGetVoumeListSelector= 20,/* DataHGetVolumeList */
    kDataWriteSelector      = 21,   /* DataHWrite */
    kDataPreextendSelector= 22, /* DataHPreextend */
    kDataSetFileSizeSelector= 23,/* DataHSetFileSize */
    kDataGetFreeSpaceSelector= 24,/* DataHGetFreeSpace */
    kDataCreateFileSelector= 25,/* DataHCreateFile */
    kDataGetPreferredBlockSizeSelector= 26,/* DataHGetPreferredBlockSize
*/
    kDataGetDeviceIndexSelector= 27,/* DataHGetDeviceIndex */
    /* 28 and 29 unused */
    kDataGetScheduleAheadTimeSelector= 30,/*

DataHGetScheduleAheadTime */
    kDataSetOSFileRefSelector= 516,/* DataHSetOSFileRef */
    kDataGetOSFileRefSelector= 517,/* DataHGetOSFileRef */

    kDataPlaybackHintsSelector= 3+0x100/* DataHPlaybackHints */
};
```

A Sample Data Handler Component

This section provides sample code for a Macintosh data handler component.

While data handler components to be used with QuickTime for Windows are functionally quite similar to Macintosh data handlers, there are some differences. QuickTime for Windows does not support a write data path. Therefore, your data handler needs to support only those functions that allow QuickTime to read movie data. In addition, Windows components are built as special dynamic link libraries (DLLs). You need to structure your code appropriately. For more information and a sample Windows data handler, see *Creating Custom Components: QuickTime for Windows*.

Listing 16-1 Sample Macintosh Data Handler

```
#include <Aliases.h>
#include <Files.h>
#include <OSUtils.h>

#include “DataHandlerPrototypes.h”

//  these selectors belong in the header file

enum {DataGetDataSelector = 2 };
enum {DataPutDataSelector = 3 };
enum {DataOpenForWriteSelector = 5 };
enum {DataCloseForWriteSelector = 6 };
enum {DataOpenForReadSelector = 8 };
enum {DataCloseForReadSelector = 9 };
enum {DataSetAliasSelector = 10 };
enum {DataGetAliasSelector = 11 };
enum {DataCompareAliasSelector = 12 };
enum {DataTaskSelector = 13 };
enum {DataScheduleDataSelector = 14 };
enum {DataCanUseDataRef = 19 };
enum {DataGetVolumeListSelector = 20 };

//  data structures

typedef struct {
    ComponentInstance self;

    AliasHandle alias;

    short           readFref;
    short           writeFref;
} DataHandlerGlobalsRecord, *DataHandlerGlobals;

//  function declarations

pascal ComponentResult main(ComponentParameters *params,
                                        Handle storage);

ComponentFunctionUPP DHSelectorLookup(short selector);
```

```
pascal ComponentResult DHOpen(DataHandlerGlobals storage,
                                    ComponentInstance self);
pascal ComponentResult DHClose(DataHandlerGlobals storage,
                                    ComponentInstance self);
pascal ComponentResult DHCanDo(DataHandlerGlobals storage,
                                    short functionSelector);
pascal ComponentResult DHVersion( DataHandlerGlobals storage);

pascal ComponentResult DHGetData(DataHandlerGlobals storage, Handle h,
                                    long offsetIntoHandle, long
offset,
                                    long size);
pascal ComponentResult DHPutData(DataHandlerGlobals storage, Handle h,
                                    long hOffset, long *offset, long
size);

pascal ComponentResult DHSetAlias(DataHandlerGlobals storage,
                                    AliasHandle alias);
pascal ComponentResult DHGetAlias(DataHandlerGlobals storage,
                                    AliasHandle *alias);
pascal ComponentResult DHCompareAlias(DataHandlerGlobals storage,
                                    AliasHandle alias, Boolean
*equal);

pascal ComponentResult DHScheduleData (DataHandlerGlobals storage,
                                    Ptr dataPtr,long fileOffset,
                                    long dataSize, long refCon,
                                    TimeRecord *timeNeededBy,
                                    DataHCompletionUPP
completionRoutine);

pascal ComponentResult DHOpenForRead(DataHandlerGlobals storage);
pascal ComponentResult DHCloseForRead(DataHandlerGlobals storage);
pascal ComponentResult DHOpenForWrite(DataHandlerGlobals storage);
pascal ComponentResult DHCloseForWrite(DataHandlerGlobals storage);

pascal ComponentResult DHGetVolumeList(DataHandlerGlobals storage,
                                    DataHVolumeList *volumeList);
pascal ComponentResult DHCanUseDataRef(DataHandlerGlobals storage,
                                    Handle dataRef, long *useFlags);
```

```
// main function

pascal ComponentResult main(ComponentParameters *params, Handle storage)
{
    ComponentResult err;
    ComponentFunctionUPP componentProc;

    componentProc = DHSelectorLookup(params->what);

    if (componentProc)
        err = CallComponentFunctionWithStorage(storage, params,

componentProc);
    else
        err = badComponentSelector;

    return err;
}

//  determine function based on selected request

ComponentFunctionUPP DHSelectorLookup(short selector)
{
    ComponentFunctionUPPcomponentProc = 0;

    switch (selector) {
        case kComponentVersionSelect:
            componentProc = (ComponentFunctionUPP)DHVersion;
            break;
        case kComponentCanDoSelect:
            componentProc = (ComponentFunctionUPP)DHCanDo;
            break;
        case kComponentCloseSelect:
            componentProc = (ComponentFunctionUPP)DHClose;
            break;
        case kComponentOpenSelect:
            componentProc = (ComponentFunctionUPP)DHOpen;
            break;

        case DataGetDataSelector:
```

```
            componentProc = (ComponentFunctionUPP)DHGetData;
            break;
        case DataPutDataSelector:
            componentProc = (ComponentFunctionUPP)DHPutData;
            break;
        case DataOpenForReadSelector:
            componentProc = (ComponentFunctionUPP)DHOpenForRead;
            break;
        case DataCloseForReadSelector:
            componentProc = (ComponentFunctionUPP)DHCloseForRead;
            break;
        case DataOpenForWriteSelector:
            componentProc = (ComponentFunctionUPP)DHOpenForWrite;
            break;
        case DataCloseForWriteSelector:
            componentProc = (ComponentFunctionUPP)DHCloseForWrite;
            break;
        case DataSetAliasSelector:
            componentProc = (ComponentFunctionUPP)DHSetAlias;
            break;
        case DataGetAliasSelector:
            componentProc = (ComponentFunctionUPP)DHGetAlias;
            break;
        case DataCompareAliasSelector:
            componentProc = (ComponentFunctionUPP)DHCompareAlias;
            break;
        case DataScheduleDataSelector:
            componentProc = (ComponentFunctionUPP)DHScheduleData;
            break;
        case DataCanUseDataRef:
            componentProc = (ComponentFunctionUPP)DHCanUseDataRef;
            break;
        case DataGetVolumeListSelector:
            componentProc = (ComponentFunctionUPP)DHGetVolumeList;
            break;
    }

    return componentProc;
}

//  open data handler connection
```

```
pascal ComponentResult DHOpen(DataHandlerGlobals storage,
                                        ComponentInstance self)
{
    ComponentResult err;

    storage =
(DataHandlerGlobals)NewPtrClear(sizeof(DataHandlerGlobalsRecord));
    if (err = MemError())
        return err;

    storage->self = (ComponentInstance)self;

    SetComponentInstanceStorage(storage->self,(Handle)storage);

    return noErr;
}

//  close component connection

pascal ComponentResult DHClose(DataHandlerGlobals storage,
                                            ComponentInstance self)
{
    if (storage != nil) {
        DHCloseForRead(storage);
        DHCloseForWrite(storage);

        if (storage->alias != nil)
            DisposeHandle((Handle)storage->alias);

        DisposePtr((Ptr)storage);
    }

    return noErr;
}

//  determine whether data handler supports request

pascal ComponentResult DHCanDo(DataHandlerGlobals storage,
                                            short functionSelector)
{
```

```
    return DHSelectorLookup(functionSelector) != 0;
}

//  return component's version

pascal ComponentResult DHVersion(DataHandlerGlobals storage)
{
    return 0x00020001;
}

//  read data

pascal ComponentResult DHGetData(DataHandlerGlobals storage, Handle h,
                                        long offsetIntoHandle, long
offset, long size)
{
    OSErr       err;
    SignedBytesaveState;

    if (!storage->readFref) {
        err = DHOpenForRead(storage);
        if (err != noErr)
            return err;
    }

    saveState = HGetState(h);
    HLock(h);
    err = DHScheduleData(storage, *h + offsetIntoHandle,
                                    offset, size, 0, nil, nil);
    HSetState(h, saveState);

    return err;
}

//  write data

pascal ComponentResult DHPutData(DataHandlerGlobals storage, Handle h,
                                        long hOffset, long *offset,
long size)
{
    OSErr       err;
```

```
    if (!storage->writeFref) {
        err = DHOpenForWrite(storage);
        if (err != noErr)
            return err;
    }

    err = SetFPos(storage->writeFref, fsFromLEOF, 0);
    if (err == noErr) {
        if (offset)
            err = GetFPos(storage->writeFref, offset);
        if (err == noErr)
            err = FSWrite(storage->writeFref, &size, *h + hOffset);
    }

    return err;
}

//  set alias

pascal ComponentResult DHSetAlias(DataHandlerGlobals storage,
                                                AliasHandle alias)
{
    OSErr err = noErr;

    // throw away the old one
    if (storage->alias) {
        DisposeHandle((Handle)storage->alias);
        storage->alias = nil;
    }

    // copy the new one, if there is one
    if (alias) {
        err = HandToHand((Handle *)&alias);
        if (err == noErr)
            storage->alias = alias;
    }

    return err;
}
```

```
//  retrieve alias

pascal ComponentResult DHGetAlias(DataHandlerGlobals storage,
                                                    AliasHandle *alias)
{
    OSErr err = noErr;

    *alias = nil;
    if (storage->alias) {
        *alias = storage->alias;
        err = HandToHand((Handle *)alias);
    }

    return err;
}

//  compare two aliases

pascal ComponentResult DHCompareAlias(DataHandlerGlobals storage,
                                                      AliasHandle alias,
Boolean *equal)
{
    OSErr err = paramErr;
    FSSpec fss1, fss2;
    Boolean whoCares;

    *equal = false;

    if (storage->alias && alias) {
        err = ResolveAlias(nil, storage->alias, &fss1, &whoCares);
        if (err == noErr) {
            err = ResolveAlias(nil, alias, &fss2, &whoCares);
            if (err == noErr) {
                *equal =(fss1.vRefNum == fss2.vRefNum) &&
                            (fss1.parID == fss2.parID) &&
                            EqualString(fss1.name, fss2.name, false,
false);
            }
        }
    }
```

```
    return err;
}

//  scheduled read

pascal ComponentResult DHScheduleData(DataHandlerGlobals storage,
                                                    Ptr dataPtr,long
fileOffset,
                                                    long dataSize, long
refCon,
                                                    TimeRecord
*timeNeededBy,
                                                    DataHCompletionUPP
completionRoutine)
{
    OSErr err;

    if (storage->readFref == 0) {
        err = DHOpenForRead(storage);
        if (err)
            return err;
    }

    err = SetFPos(storage->readFref, fsFromStart, fileOffset);
    if (err == noErr)
        err = FSRead(storage->readFref, &dataSize, dataPtr);

    // Always call completion routine, even on an error.
    if (completionRoutine != nil)
        (*completionRoutine)(dataPtr, refCon, err);

    return err;
}

//  open container for read

pascal ComponentResult DHOpenForRead(DataHandlerGlobals storage)
{
    OSErr err;
    FSSpec fss;
    Boolean whoCares;
```

```
    if (storage->readFref != 0)
        return noErr;

    if (storage->alias == nil)
        return dataNoDataRef;

    err = ResolveAlias(nil, storage->alias, &fss, &whoCares);
    if (err) return err;

    err = FSpOpenDF(&fss, fsRdPerm, &storage->readFref);

    return err;
}

//  close container after reading

pascal ComponentResult DHCloseForRead(DataHandlerGlobals storage)
{
    if (storage->readFref) {
        FSClose(storage->readFref);
        storage->readFref = 0;
    }

    return noErr;
}

//  open container for write

pascal ComponentResult DHOpenForWrite(DataHandlerGlobals storage)
{
    OSErr err;
    FSSpec fss;
    Boolean whoCares;

    if (storage->writeFref != 0)
        return noErr;

    if (storage->alias == nil)
        return dataNoDataRef;
```

```
    err = ResolveAlias(nil, storage->alias, &fss, &whoCares);
    if (err) return err;

    err = FSpOpenDF(&fss, fsRdWrPerm, &storage->writeFref);

    return err;
}

//  close container after writing

pascal ComponentResult DHCloseForWrite(DataHandlerGlobals storage)
{
    if (storage->writeFref) {
        FSClose(storage->writeFref);
        storage->writeFref = 0;
    }

    return noErr;
}

//
//  This function limits the set of drives this data handler will be
used to
//  read from to those with names beginning with the letter Q.
//
Boolean isVRefNumOK(short vRefNum);
Boolean isVRefNumOK(short vRefNum)
{
    ParamBlockRec pb;
    Str63 name;

    name[0] = 0;
    pb.volumeParam.ioVolIndex = 0;
    pb.volumeParam.ioVRefNum = vRefNum;
    pb.volumeParam.ioNamePtr = name;
    if (PBGetVInfoSync(&pb) != noErr)
        return false;

    return (name[1] == 'Q') || (name[1] == 'q');
}
```

```
//  determine whether we can handle the data reference

pascal ComponentResult DHCanUseDataRef(DataHandlerGlobals storage,
                                                    Handle dataRef, long
*useFlags)
{
    OSErr err;
    FSSpec fss;
    Boolean whoCares;

    *useFlags = 0;

    err = ResolveAlias(nil, (AliasHandle)dataRef, &fss, &whoCares);
    if (err) return err;

    if (isVRefNumOK(fss.vRefNum))
        *useFlags = kDataHCanRead | kDataHSpecialRead | kDataHCanWrite;

    return noErr;
}

//
//  This call is only required for data handlers with a subtype of
//  rAliasType ('alis').
//
pascal ComponentResult DHGetVolumeList(DataHandlerGlobals storage,
                                                    DataHVolumeList
*volumeList)
{
    OSErr err = noErr;
    DataHVolumeList list;
    VCB *vq;

    list = (DataHVolumeList)NewHandle(0);
    if (err = MemError())
        goto bail;

    vq = (VCB *)GetVCBQHdr()->qHead;
    while (vq) {
        if (isVRefNumOK(vq->vcbVRefNum)) {
            DataHVolumeListRecord vlr;
```

```
            // add it to our list
            vlr.vRefNum = vq->vcbVRefNum;
            vlr.flags = kDataHCanRead | kDataHSpecialRead |
kDataHCanWrite;
            err = PtrAndHand((Ptr)&vlr, (Handle)list, sizeof(vlr));
            if (err)
                goto bail;
        }
        vq = (VCB *)vq->qLink;
    }

bail:
    if (err) {
        DisposeHandle((Handle)list);
        list = nil;
    }
    *volumeList = list;
    return err;
}
```

Data Handler Components Reference

This section describes the functions your data handler component may support. Some of these functions are optional—your component should support only those functions that are appropriate to it.

Data Handler Components Functions

This section describes the functions that may be supported by data handler components and is divided into the following topics:

- "Selecting a Data Handler" (page 16-29) describes the functions that allow client programs, such as the Movie Toolbox, to select an appropriate data handler for a data reference.

- "Working With Data References" (page 16-36) describes the functions that allow client programs to manage a data handler's current data reference.
- "Reading Movie Data" (page 16-41) describes the functions that allow client programs to retrieve data from a data handler.
- "Writing Movie Data" (page 16-50) describes the functions that allow client programs to store data using a data handler.
- "Managing Data Handler Components" (page 16-58) describes the functions that allow client programs to manage their interactions with data handler components.
- "Completion Function" (page 16-61) describes the interface that must be provided by a client program's data-handler completion function.

Selecting a Data Handler

In order for client programs to choose the best data handler component for a data reference, Apple has defined some functions that allow applications to interrogate a data handler's capabilities.

The `DataHGetVolumeList` function allows an application to obtain a list of the volumes your data handler can support. The `DataHCanUseDataRef` function allows your data handler to examine a specific data reference and indicate its ability to work with the associated container. The `DataHGetDeviceIndex` function allows applications to determine whether different data references identify containers that reside on the same device.

By way of illustration, the Movie Toolbox uses the `DataHGetVolumeList` and `DataHCanUseDataRef` functions as follows. During startup, and whenever a new volume is mounted, the Movie Toolbox calls each data handler's `DataHGetVolumeList` function in order to obtain information about each handler's general capabilities. Specifically, the Movie Toolbox calls each component's `GetDataHandler`, `DataHGetVolumeList`, and `CloseComponent` functions.

Whenever an application opens a movie, the Movie Toolbox selects the best data handler for the movie's container. This may involve calling each appropriate data handler's `DataHCanUseDataRef` function (in some cases, a data handler may indicate that it does not need to examine a data reference before accessing it—see the description of the `DataHGetVolumeList` function for more information). For each data handler that can support the data reference (that is, has the correct component subtype value) and needs to be interrogated, the

Movie Toolbox calls the component's `GetDataHandler`, `DataHCanUseDataRef`, and `CloseComponent` functions. Based on the resulting information, the Movie Toolbox selects the best data handler for the application.

For more information on selecting a data handler, see "Selecting a Data Handler" (page 16-8).

DataHGetVolumeList

In response to the `DataHGetVolumeList` function, your data handler component returns a list of the volumes your component can access, along with flags indicating your component's capabilities for each volume.

```
pascal ComponentResult DataHGetVolumeList (DataHandler dh,
                    DataHVolumeList *volumeList);
```

`dh` Identifies the calling program's connection to your data handler component.

`volumeList` Contains a pointer to a field that your data handler component uses to return a handle to a volume list. Your component constructs the volume list by allocating a handle and filling it with a series of `DataHVolumeListRecord` structures (one structure for each volume your component can access). This structure is described later in this section.

DISCUSSION

In order to reduce the delay that may result from choosing an appropriate data handler for a volume, the Movie Toolbox maintains a list of data handlers and the volumes they support. The Movie Toolbox uses the `DataHGetVolumeList` function to build that list.

When your component receives this function, it should scan the available volumes and create a series of `DataHVolumeListRecord` structures—one structure

for each volume your component can access. This structure is defined as follows:

```
typedef struct DataHVolumeListRecord {
                    shortvRefNum;/* reference number */
                    longflags;/* capability flags */
                    } DataHVolumeListRecord, *DataHVolumeListPtr,
                    **DataHVolumeList;
```

`vRefNum` Contains the volume reference number assigned to the volume.

`flags` Indicates the level of support your data handler can provide for this volume. These flags are similar to those defined for the `DataHCanUseDataRef` function, though there is one additional flag. Your component should set every appropriate flag to 1 (set unused flags to 0).

`kDataHCanRead`
Indicates that your data handler can read from the volume.

`kDataHSpecialRead`
Indicates that your data handler can read from the volume using a specialized method. For example, your data handler might support access to networked multimedia servers using a special protocol. In that case, your component would set this flag to 1 whenever the volume resides on a supported server.

`kDataHSpecialReadFile`
Reserved for use by Apple.

`kDataHCanWrite`
Indicates that your data handler can write data to the volume. In particular, use this flag to indicate that your data handler's `DataHPutData` function will work with this volume.

`kDataHSpecialWrite`
Indicates that your data handler can write to the volume using a specialized method. As with the `kDataHSpecialRead` flag, your data handler would use this flag to indicate that your

component can access the volume using specialized support (for example, special network protocols).

`kDataHCanStreamingWrite`
Indicates that your data handler can support the special write functions for capturing movie data when writing to this volume. These functions are described in"Writing Movie Data" (page 16-50) " of this chapter.

`kDataHMustCheckDataRef`
Instructs the calling program that your component must check each data reference before it can accurately report its capabilities. If you set this flag to 1, the Movie Toolbox will call your component's `DataHCanUseDataRef` function before it assigns a container to your data handler. Note, however, that this may slow the data handler selection process somewhat.

Your data handler may use any facilities necessary to determine whether it can access the volume, including opening a container on the volume. Your component should set to 1 as many of the capability flags as are appropriate for each volume. Do not include records for volumes your handler cannot support.

For example, if your component supports networked multimedia servers using a special set of protocols, your data handler should set the `kDataHCanRead` and `kDataHCanSpecialRead` flags to 1 for any volume that is on that server. In addition, if your component can write to a volume on the server, set the `kDataHCanWrite` and `kDataHCanSpecialWrite` flags to 1 (perhaps along with `kDataHCanStreamingWrite`). However, your component should create entries only for those volumes that support your protocols.

It is the calling program's responsibility to dispose of the handle returned by your component.

The Movie Toolbox tracks mounting and unmounting removable volumes, and keeps its volume list current. As a result, the Movie Toolbox may call your component's `DataHGetVolumeList` function whenever a removable volume is mounted.

If your data handler does not process data that is stored in file system volumes, you need not support this function.

RESULT CODES

Memory Manager errors

DataHCanUseDataRef

The `DataHCanUseDataRef` function allows your data handler to report whether it can access the data associated with a specified data reference.

```
pascal ComponentResult DataHCanUseDataRef (DataHandler dh, Handle
                    dataRef, long *useFlags);
```

`dh` Identifies the calling program's connection to your data handler component.

`dataRef` Specifies the data reference. This parameter contains a handle to the information that identifies the container in question.

`useFlags` Contains a pointer to a field that your data handler component uses to indicate its ability to access the container identified by the `dataRef` parameter. Your data handler may use the following flags (set all flags that are appropriate to 1; set unused flags to 0):

`kDataHCanRead`
Indicates that your data handler can read from the container.

`kDataHSpecialRead`
Indicates that your data handler can read from the container using a specialized method. For example, your data handler might support access to networked multimedia servers using a special protocol. In that case, your component would set this flag to 1 whenever the data reference identifies a container on a supported server.

`kDataHSpecialReadFile`
Indicates that your data handler can read from the container using a specialized method that is particular to the type of container in question.

For example, your data handler may use a different method for some types of containers (say, a Hypercard stack).

This flag represents a special case of the `kDataHSpecialRead` flag. That is, this flag is appropriate only if you have also set `kDataHSpecialRead` to 1.

`kDataHCanWrite`

Indicates that your data handler can write data to the container. In particular, use this flag to indicate that your data handler's `DataHPutData` function will work with this data reference.

`kDataHSpecialWrite`

Indicates that your data handler can write to the container using a specialized method. As with the `kDataHSpecialRead` flag, your data handler would use this flag to indicate that the data reference identifies a container which your component can access using specialized support (for example, special network protocols).

`kDataHCanStreamingWrite`

Indicates that your data handler can support the special write functions for capturing movie data when writing to this container. These functions are described later in this chapter, in "Writing Movie Data."

If your data handler cannot access the container, set the field to 0.

DISCUSSION

Apple's standard data handler sets both the `kDataHCanRead` and `kDataHCanWrite` flags to 1 for any data reference it receives, indicating that it can read from and write to any volume.

Your component should set to 1 as many of the capability flags as are appropriate for the specified data reference. Conversely, be sure to set the flags to 0 if your component cannot support the container. For example, if your component supports networked multimedia servers using a special set of

protocols, your data handler should set the `kDataHCanRead` and `kDataHCanSpecialRead` flags to 1 for any container that is on that server. In addition, if your component can write to the server, set the `kDataHCanWrite` and `kDataHCanSpecialWrite` flags to 1 (perhaps along with `kDataHCanStreamingWrite`). However, your component should set the flags field to 0 for any container that is not on a server that supports your protocols.

Your data handler may use any facilities necessary to determine whether it can access the container. Bear in mind, though, that your component should try to be as quick about this determination as possible, in order to minimize the chance that the delay will be noticed by the user.

SEE ALSO

The Movie Toolbox calls your component's `DataHGetVolumeList` function to retrieve your data handler's capabilities for an entire volume.

DataHGetDeviceIndex

In response to the `DataHGetDeviceIndex` function, your data handler component returns a value that identifies the device on which a data reference resides.

```
pascal ComponentResult DataHGetDeviceIndex (DataHandler dh, long
                    *deviceIndex);
```

`dh`	Identifies the calling program's connection to your data handler component.
`deviceIndex`	Contains a pointer to a field that your data handler component uses to return a device identifier value.

DISCUSSION

Some client programs may need to account for the fact that two or more data references reside on the same device. For instance, this may affect storage-allocation requirements. This function allows such client programs to obtain this information from your data handler.

Your component may use any identifier value that is appropriate (as an example, Apple's HFS data handler uses the volume reference number). The

client program should do nothing with the value other than compare it with other identifiers returned by your data handler component.

Working With Data References

All data handler components use data references to identify and locate a movie's container. Different types of containers may require different types of data references. For example, a reference to a memory-based movie may be a handle, while a reference to a file-based movie may be an alias.

Client programs can correlate data references with data handlers by matching the component's subtype value with the data reference type—the subtype value indicates the type of data reference the component supports. All data handlers with the same subtype value must support the same data reference type. To continue the previous example, Apple's memory-based data handler for the Macintosh uses handles (and has a subtype value of `'hndl'`), while the HFS data handler uses Alias Manager aliases (its subtype value is `'alis'`).

The `DataHSetDataRef` and `DataHGetDataRef` functions allow applications to assign your data handler's current data reference. The `DataHCompareDataRef` function asks your component to compare a data reference against the current data reference and indicate whether the references are equivalent (that is, refer to the same container). The `DataHResolveDataRef` permits your component to locate a data reference's container.

The `DataHSetOSFileRef` and `DataHGetOSFileRef` functions provide an alternative, system-specific mechanism for assigning your data handler's current data reference.

For more information on data references, see "Managing Data References" (page 16-10).

DataHSetDataRef

The `DataHSetDataRef` function assigns a data reference to your data handler component.

```
pascal ComponentResult DataHSetDataRef (DataHandler dh, Handle dataRef);
```

dh	Identifies the calling program's connection to your data handler component.
dataRef	Specifies the data reference. This parameter contains a handle to the information that identifies the container in question. Your component must make a copy of this handle.

DISCUSSION

Note that the type of data reference always corresponds to the type that your component supports, and that you specify in the component subtype value of your data handler. As a result, the client program does not provide a data reference type value (unlike the Movie Toolbox's data reference functions).

The client program is responsible for disposing of the handle. Consequently, your component must make a copy of the data reference handle.

RESULT CODES

Memory Manager errors

DataHGetDataRef

The `DataHGetDataRef` function retrieves your component's current data reference.

```
pascal ComponentResult DataHGetDataRef (DataHandler dh, Handle *dataRef);
```

dh	Identifies the calling program's connection to your data handler component.
dataRef	Contains a pointer to a data reference handle. Your component should make a copy of its current data reference in a handle and return that handle in this field. The client program is responsible for disposing of that handle.

RESULT CODES

Memory Manager errors

DataHCompareDataRef

Your component compares a supplied data reference against its current data reference and returns a Boolean value indicating whether the data references are equivalent (that is, the two data references identify the same container).

```
pascal ComponentResult DataHCompareDataRef (DataHandler dh, Handle
                    dataRef, Boolean *equal);
```

`dh` Identifies the calling program's connection to your data handler component.

`dataRef` Specifies the data reference to be compared to your component's current data reference.

`equal` Contains a pointer to a Boolean. Your component should set that Boolean to `true` if the two data references identify the same container. Otherwise, set the Boolean to `false`.

DISCUSSION

Note that your component cannot simply compare the bits in the two data references. For example, two completely different aliases may refer to the same HFS file. Consequently, you need to completely resolve the data reference in order to determine the file identified by the reference.

DataHResolveDataRef

The `DataHResolveDataRef` function instructs your data handler component to locate the container associated with a given data reference.

```
pascal ComponentResult DataHResolveDataRef (DataHandler dh, Handle
                    theDataRef, Boolean *wasChanged,
                    Boolean userInterfaceAllowed);
```

`dh` Identifies the calling program's connection to your data handler component.

`theDataRef` Specifies the data reference to be resolved.

`wasChanged` Contains a pointer to a Boolean. Your component should set that Boolean to `true` if, in locating the container, your data handler updates any information in the data reference.

`userInterfaceAllowed`
Indicates whether your component may interact with the user when locating the container. If this parameter is set to `true`, your component may ask the user to help locate the container (for instance, by presenting a Find File dialog box).

DISCUSSION

This function is, essentially, equivalent to the Alias Manager's `ResolveAlias` function. The client program asks your component to locate the container that is associated with a given data reference. If your component determines that the data reference needs to be updated with more accurate location information, it should put the new information in the supplied data reference (and set the Boolean referred to by the `wasChanged` parameter to `true`).

Client programs may call your data handler's `DataHResolveDataRef` function at any time. Typically, however, the Movie Toolbox uses this function as part of its strategy for opening and reading a movie container. As such, you can expect that the supplied data reference will identify a container that your component can support.

DataHSetOSFileRef

The `DataHSetOSFileRef` function assigns a movie container to your data handler component. Applications may use this function instead of calling the `DataHSetDataRef` function in cases where the applications have already opened the container.

```
pascal ComponentResult DataHSetOSFileRef (DataHandler dh, long ref, long
                    flags);
```

`dh` Identifies the calling program's connection to your data handler component.

`ref`	Specifies the container. This parameter contains an operating system-specific file-access token. For example, on the Macintosh an application would supply the file reference it obtained by calling the `FSOpenFile` function. Under Windows, this parameter would contain an `HFILE` value obtained from the `OpenFile` function.
`flags`	Specifies access flags for the container. This parameter contains the access flags the application used when opening the container. Again, these are operating system-specific.

DISCUSSION

This function provides an alternative mechanism for assigning your data handler's current container. In some cases, an application may have created or opened a movie container prior to assigning the container to your handler. In such cases, the application may choose to provide its access token to your data handler, rather than using the `DataHSetDataRef` function to assign a data reference. The application must have opened the file before calling this function.

Note that your data handler must implement this function in a system-specific manner, and must verify that the access token is valid.

Applications must still call your handlers `DataHOpenForRead` or `DataHOpenForWrite` functions, as appropriate, before using your data handler to access the container.

RESULT CODES

`invalidDataRef`	–2012	Application already set a data reference
`memFullErr`	–108	Insufficient memory for operation

DataHGetOSFileRef

The `DataHGetOSFileRef` function retrieves your component's container access token, if it was assigned using the `DataHSetOSFileRef` function.

```
pascal ComponentResult DataHGetOSFileRef (DataHandler dh, long *ref,
                    long *flags);
```

dh	Identifies the calling program's connection to your data handler component.
ref	Contains a pointer to a long. Your component should return the container access token that the application provided when it called your DataHSetOSFileRef function.
flags	Contains a pointer to a long. Your component should return the access flags that the application provided when it called your DataHSetOSFileRef function.

RESULT CODES

invalidDataRef	–2012	Application already set a data reference
memFullErr	–108	Insufficient memory for operation

Reading Movie Data

Data handler components provide two basic read facilities. The DataHGetData function is a fully synchronous read operation, while the DataHScheduleData function is asynchronous. Applications provide scheduling information when they call your component's DataHScheduleData function. When your component processes the queued request, it calls the application's data-handler completion function (for more information, see "Completion Function" (page 16-61) later in this chapter). By calling your component's DataHFinishData function, applications can force your component to process queued read requests. Applications may call your component's DataHGetScheduleAheadTime function in order to determine how far in advance your component prefers to get read requests.

Before any application can read data from a data reference, it must open read access to that reference by calling your component's DataHOpenForRead function. The DataHCloseForRead function closes that read access path.

For more information on reading movie data, see "Retrieving Movie Data" (page 16-11).

DataHOpenForRead

Your component opens its current data reference for read-only access.

```
pascal ComponentResult DataHOpenForRead (DataHandler dh);
```

`dh` Identifies the calling program's connection to your data handler component.

DISCUSSION

After setting your component's current data reference by calling the `DataHSetDataRef` function, client programs call the `DataHOpenForRead` function in order to start reading from the data reference. Your component should open the data reference for read-only access. If the data reference is already open or cannot be opened, return an appropriate error code.

Note that the Movie Toolbox may try to read data from a data reference without calling your component's `DataHOpenForRead` function. If this happens, your component should open the data reference for read-only access, respond to the read request, and then leave the data reference open in anticipation of later read requests.

DataHCloseForRead

Your component closes read-only access to its data reference.

```
pascal ComponentResult DataHCloseForRead (DataHandler dh);
```

`dh` Identifies the calling program's connection to your data handler component.

DISCUSSION

Note that a client program may close its connection to your component (by calling the Component Manager's `CloseComponent` function) without closing the read path. If this happens, your component should close the data reference before closing the connection.

RESULT CODES

`dataNotOpenForRead`	–2042	Data reference not open for read
`dataAlreadyClosed`	–2045	This reference already closed

DataHGetData

Your component reads data from its current data reference. This is a synchronous read operation.

```
pascal ComponentResult DataHGetData (DataHandler dh, Handle h, long
                    hOffset, long offset, long size);
```

`dh`	Identifies the calling program's connection to your data handler component.
`h`	Specifies the handle to receive the data.
`hOffset`	Identifies the offset into the handle where your component should return the data.
`offset`	Specifies the offset in the data reference from which your component is to read.
`size`	Specifies the number of bytes to read.

DISCUSSION

The `DataHGetData` function provides a high-level read interface. This is a synchronous read operation; that is, the client program's execution is blocked until your component returns control from this function. As a result, most time-critical clients use the `DataHScheduleData` function to read data.

Note that the Movie Toolbox may try to read data from a data reference without calling your component's `DataHOpenForRead` function. If this happens, your component should open the data reference for read-only access, respond to the read request, and then leave the data reference open in anticipation of later read requests.

SEE ALSO

Client programs can force your component to invalidate any cached data by calling your component's `DataHFlushCache` function.

DataHScheduleData

Your component reads data from its current data reference. This can be a synchronous read operation or an asynchronous read operation.

```
pascal ComponentResult DataHScheduleData (DataHandler dh,
                    Ptr placeToPutDataPtr, long fileOffset, long
                    dataSize, long refCon,
                    DataHSchedulePtr scheduleRec, DHCompleteProc
                    completionRtn);
```

`dh` Identifies the calling program's connection to your data handler component.

`placeToPutDataPtr`
Specifies the location in memory that is to receive the data.

`fileOffset` Specifies the offset in the data reference from which your component is to read.

`dataSize` Specifies the number of bytes to read.

`refCon` Contains a reference constant that your data handler component should provide to the data-handler completion function specified with the `completionRtn` parameter.

`scheduleRec` Contains a pointer to a schedule record. If this parameter is set to `nil`, then the client program is requesting a synchronous read operation (that is, your data handler must return the data before returning control to the client program).

If this parameter is not set to `nil`, it must contain the location of a schedule record that has timing information for an asynchronous read request. Your data handler should return control to the client program immediately, and then call the client's data-handler completion function when the data is ready. The schedule record is described later in this section.

`completionRtn`

Contains a pointer to a data-handler completion function. When your data handler finishes with the client program's read request, your component must call this routine. Be sure to call this routine even if the request fails. Your component should pass the reference constant that the client program provided with the `refCon` parameter.

The client program must provide a completion routine for all asynchronous read requests (that is, all requests that include a valid schedule record). For synchronous requests, client programs should set this parameter to `nil`. However, if the function is provided, your handler must call it, even after synchronous requests.

DISCUSSION

The `DataHScheduleData` function provides both a synchronous and an asynchronous read interface. Synchronous read operations work like the `DataHGetData` function—the data handler component returns control to the client program only after it has serviced the read request. Asynchronous read operations allow client programs to schedule read requests in the context of a specified QuickTime time base. Your data handler queues the request and immediately returns control to the calling program. After your component actually reads the data, it calls the client program's data-handler completion function.

If your component cannot satisfy the request (for example, the request requires data more quickly than you can deliver it), your component should reject the request immediately, rather than queuing the request and then calling the client's data-handler completion function.

The client program provides scheduling information for scheduled reads in a schedule record. This structure is defined as follows:

```
typedef struct DataHScheduleRecord {
                    TimeRecordtimeNeededBy;/* schedule info */
                    longextendedID;/* type of data */
                    longextendedVers;/* reserved */
                    Fixedpriority;/* priority */
                    } DataHScheduleRecord, *DataHSchedulePtr;
```

`timeNeededBy` Specifies the time at which your data handler must deliver the requested data to the calling program. This time value is relative to the time base that is contained in this time record.

During pre-roll operations, the Movie Toolbox may use special values in certain time record fields. The time record fields in question are the `scale` and `value` fields. By correctly interpreting the values of these fields, your data handler can queue up the pre-roll read requests in the most efficient way for its device.

There are two types of pre-roll read operations. The first type is a required read; that is, the Movie Toolbox requires that the read operation be satisfied before the movie starts playing. The second type is an optional read. If your data handler can satisfy the read operation as part of the pre-roll operation, it should do so. Otherwise, your data handler may satisfy the request at a specified time while the movie is playing.

The Movie Toolbox indicates that a pre-roll read request is required by setting the `scale` field of the time record to –1. This literally means that the request is scheduled for a time that is infinitely far into the future. Your data handler should collect all such read requests, order them most efficiently for your device, and process them when the Movie Toolbox calls your component's `DataHFinishData` function.

For optional pre-roll read requests, the Movie Toolbox sets the `scale` field properly, but negates the contents of the `value` field. Your data handler has the option of delivering the data for this request with the required data, if that can be done efficiently. Otherwise, your data handler may deliver the data at its schedule time. You determine the scheduled time by negating the contents of the `value` field (that is, multiplying by –1).

For more information about pre-roll operations, see "Retrieving Movie Data," earlier in this chapter.

`extendedID` Indicates the type of data that follows in the remainder of the record. The following values are valid:

`kDataHExtendedSchedule`
: The remainder of the record contains extended scheduling information.

If the `extendedID` field is set to `kDataHExtendedSchedule`, the remainder of the schedule record is defined as follows:

`extendedVers` Reserved; this field should always be set to 0.

`priority` Indicates the relative importance of the data request. Client programs assign a value of 100.0 to data requests the must be delivered. Lower values indicate relatively less critical data. If your data handler must accommodate bandwidth limitations when delivering data, your component may use this value as an indication of which requests can be dropped with the least impact on the client program.

As an example, consider using priorities in a frame-differenced movie. Key frames might have priority values of 100.0, indicating that they are essential to proper playback. As you move through the frames following a key frame, each successive frame might have a lower priority value. Once you drop a frame, you must drop all successive frames of equal or lower priority until you reach another key frame, because each of these frames would rely on the dropped one for some image data.

Note that the Movie Toolbox may try to read data from a data reference without calling your component's `DataHOpenForRead` function. If this happens, your component should open the data reference for read-only access, respond to the read request, and then leave the data reference open in anticipation of later read requests.

SEE ALSO

Client programs can force your component to invalidate any cached data by calling your component's `DataHFlushCache` function.

DataHFinishData

The `DataHFinishData` function instructs your data handler component to complete or cancel one or more queued read requests. The client program

would have issued those read requests by calling your component's DataHScheduleData function.

```
pascal ComponentResult DataHFinishData (DataHandler dh,
                    Ptr placeToPutDataPtr, Boolean cancel);
```

dh — Identifies the calling program's connection to your data handler component.

placeToPutDataPtr — Specifies the location in memory that is to receive the data. The value of this parameter identifies the specific read request to be completed. If this parameter is set to nil, the call affects all pending read requests.

cancel — Indicates whether the calling program wants to cancel the outstanding request. If this parameter is set to true, your data handler should cancel the request (or requests) identified by the placeToPutDataPtr parameter.

DISCUSSION

Client programs use the DataHFinishData function either to cancel outstanding read requests or to demand that the requests be serviced immediately. Pre-roll operations are a special case of the immediate service request. The client program will have queued one or more read requests with their scheduled time of delivery set infinitely far into the future. Your data handler queues those requests until the client program calls the DataHFinishData function demanding that all outstanding read requests be satisfied immediately.

Note that your component must call the client program's data-handler completion function for each queued request, even though the client program called the DataHFinishData function. Be sure to call the completion function for both canceled and completed read requests.

SEE ALSO

Client programs queue read requests by calling your component's DataHScheduleData function.

DataHGetScheduleAheadTime

The `DataHGetScheduleAheadTime` function allows your data-handler component to report how far in advance it prefers clients to issue read requests.

```
pascal ComponentResult DataHGetScheduleAheadTime (DataHandler dh,
                    long *millisecs);
```

`dh` Identifies the calling program's connection to your data handler component.

`millisecs` Contains a pointer to a long. Your component should set this field with a value indicating the number of milliseconds you prefer to receive read requests in advance of the time when the data must be delivered.

DISCUSSION

This function allows your data handler to tell the client program how far in advance it should schedule its read requests. By default, the Movie Toolbox issues scheduled read requests between 1 and 2 seconds before it needs the data from those requests. For some data handlers, however, this may not be enough time. For example, some data handlers may have to accommodate network delays when processing read requests. Client programs that call this function may try to respect your component's preference.

Note, however, that not all client programs will call this function. Further, some clients may not be able to accommodate your preferred time in all cases, even if they have asked for your component's preference. As a result, your component should have a strategy for handling requests that do not provide enough advanced scheduling time. For example, if your component receives a `DataHScheduleData` request that it cannot satisfy, it can fail the request with an appropriate error code.

SEE ALSO

Client programs queue read requests by calling your component's `DataHScheduleData` function.

Writing Movie Data

As with reading movie data, data handlers provide two distinct write facilities. The `DataHPutData` function is a simple synchronous interface that allows applications to append data to the end of a container.

The `DataHWrite` function is a more capable, asynchronous write function that is suitable for movie capture operations. As is the case with the `DataHScheduleData` function, your component calls the application's data-handler completion function when you are done with the write request.

There are several other helper functions that allow applications to prepare your data handler for a movie capture operation. The `DataHCreateFile` function asks your component to create a new container. The `DataHSetFileSize` and `DataHGetFileSize` functions work with a container's size, in bytes. The `DataHGetFreeSpace` function allows applications to determine when to make a container larger. The `DataHPreextend` function asks your component to make a container larger. Applications may call your component's `DataHGetPreferredBlockSize` function in order to determine how best to interact with your data handler.

Before writing data to a data reference, applications must call your component's `DataHOpenForWrite` function to open a write path to the container. The `DataHCloseForWrite` function closes that write path.

Note that some data handlers may not support write operations. For example, some shared devices, such as a CD-ROM "jukebox", may be read-only devices. As a result, it is very important that your data handler correctly report its write capabilities to client programs. See "Selecting a Data Handler" (page 16-8) for information about the functions that client programs use to interrogate your data handler. For more information on writing movie data, see "Storing Movie Data" (page 16-12).

DataHOpenForWrite

Your component opens its current data reference for write-only access.

```
pascal ComponentResult DataHOpenForWrite (DataHandler dh);
```

`dh` Identifies the calling program's connection to your data handler component.

DISCUSSION

After setting your component's current data reference by calling the `DataHSetDataRef` function, client programs call the `DataHOpenForWrite` function in order to start writing to the data reference. Your component should open the data reference for write-only access. If the data reference is already open or cannot be opened, return an appropriate error code.

RESULT CODES

`dataAlreadyOpenForWrite`	–2044	Data reference already open for write

DataHCloseForWrite

Your component closes write-only access to its data reference.

```
pascal ComponentResult DataHCloseForWrite (DataHandler dh);
```

`dh` Identifies the calling program's connection to your data handler component.

DISCUSSION

Note that a client program may close its connection to your component (by calling the Component Manager's `CloseComponent` function) without closing the write path. If this happens, your component should close the data reference before closing the connection.

RESULT CODES

dataNotOpenForWrite	–2043	Data reference not open for write
dataAlreadyClosed	–2045	This reference already closed

DataHPutData

Your component writes data to its current data reference. This is a synchronous write operation that appends data to the end of the current data reference.

```
pascal ComponentResult DataHPutData (DataHandler dh, Handle h, long
                    hOffset, long *offset, long size);
```

dh	Identifies the calling program's connection to your data handler component.
h	Specifies the handle that contains the data to be written to the data reference.
hOffset	Identifies the offset into the handle h to the data to be written.
offset	Contains a pointer to a long. Your component returns the offset in the data reference at which your component wrote the data.
size	Specifies the number of bytes to write.

DISCUSSION

The DataHPutData function provides a high-level write interface. This is a synchronous write operation that only appends data to the end of the current data reference. That is, the client program's execution is blocked until your component returns control from this function, and the client cannot control where the data is written. As a result, most movie-capture clients (for example, Apple's sequence grabber component) use the DataHWrite function to write data when creating movies.

RESULT CODES

`dataNotOpenForWrite`	–2043	Data reference not open for write

SEE ALSO

Client programs can force your component to write any cached data by calling your component's `DataHFlushData` function.

DataHWrite

Your component writes data to its current data reference. This can be a synchronous write operation or an asynchronous operation, and can write data to any location in the container.

```
pascal ComponentResult DataHWrite (DataHandler dh, Ptr data, long
                    offset, long size, DHCompleteProc completion,
                    long refCon);
```

`dh` Identifies the calling program's connection to your data handler component.

`data` Specifies a pointer to the data to be written. Client programs should lock the memory area holding this data, allowing your component's `DataHWrite` function to move memory.

`offset` Specifies the offset (in bytes) to the location in the current data reference at which to write the data.

`size` Specifies the number of bytes to write.

`completion` Contains a pointer to a data-handler completion function. When your data handler finishes with the client program's write request, your component must call this routine. Be sure to call this routine even if the request fails. Your component should pass the reference constant that the client program provided with the `refCon` parameter.

The client program must provide a completion routine for all asynchronous write requests. For synchronous requests, client programs should set this parameter to `nil`.

refCon — Contains a reference constant that your data handler component should provide to the data-handler completion function specified with the `completion` parameter.

For synchronous operations, client programs should set this parameter to 0.

DISCUSSION

The `DataHWrite` function provides both a synchronous and an asynchronous write interface. Synchronous write operations work like the `DataHPutData` function—the data handler component returns control to the client program only after it has serviced the write request. Asynchronous write operations allow client programs to queue write requests. Your data handler queues the request and immediately returns control to the calling program. After your component actually writes the data, it calls the client program's data-handler completion function.

RESULT CODES

`dataNotOpenForWrite`	–2043	Data reference not open for write

SEE ALSO

Client programs can force your component to write any cached data by calling your component's `DataHFlushData` function.

DataHSetFileSize

Your component sets the size, in bytes, of the current data reference.

```
pascal ComponentResult DataHSetFileSize (DataHandler dh, long fileSize);
```

dh — Identifies the calling program's connection to your data handler component.

fileSize — Specifies the new size of the container corresponding to the current data reference, in bytes.

DISCUSSION

The `DataHSetFileSize` function is functionally equivalent to the File Manager's `SetEOF` function. If the client program specifies a new size that is greater than the current size, your component should extend the container to accommodate that new size. If the client program specifies a container size of 0, your component should free all of the space occupied by the container.

DataHGetFileSize

Your component returns the size, in bytes, of the current data reference.

```
pascal ComponentResult DataHGetFileSize (DataHandler dh, long *fileSize);
```

`dh` Identifies the calling program's connection to your data handler component.

`fileSize` Contains a pointer to a long. Your component returns the size of the container corresponding to the current data reference, in bytes.

DISCUSSION

The `DataHGetFileSize` function is functionally equivalent to the File Manager's `GetEOF` function.

DataHCreateFile

Your component creates a new container that meets the specifications of the current data reference.

```
pascal ComponentResult DataHCreateFile (DataHandler dh, OSType creator,
                    Boolean deleteExisting);
```

`dh` Identifies the calling program's connection to your data handler component.

`creator` Specifies the creator type of the new container. If the client program sets this parameter to 0, your component should choose a reasonable value (for example, `'TVOD'`, the creator type for Apple's movie player).

`deleteExisting`
Indicates whether to delete any existing data. If this parameter is set to `true` and a container already exists for the current data reference, your component should delete that data before creating the new container. If this parameter is set to `false`, your component should preserve any data that resides in the container defined by the current data reference (if there is any).

DataHGetPreferredBlockSize

The `DataHGetPreferredBlockSize` function allows your component to report the block size that it prefers to use when accessing the current data reference.

```
pascal ComponentResult DataHGetPreferredBlockSize (DataHandler dh,
                    long *blockSize);
```

`dh` Identifies the calling program's connection to your data handler component.

`blockSize` Contains a pointer to a long. Your component returns the size of blocks (in bytes) it prefers to use when accessing the current data reference.

DISCUSSION

Different devices use different file system block sizes. This function allows your component to report its preferred block size to the client program. Note that the client program is not required to use this block size when making requests. Some clients may, however, try to accommodate your component's preference.

DataHGetFreeSpace

Your component reports the number of bytes available on the device that contains the current data reference.

```
pascal ComponentResult DataHGetFreeSpace (DataHandler dh, unsigned
                    long *freeSize);
```

`dh` Identifies the calling program's connection to your data handler component.

`freeSize` Contains a pointer to an unsigned long. Your component returns the number of bytes of free space available on the device that contains the container referred to by the current data reference.

DataHPreextend

Your component allocates new space for the current data reference, enlarging the container.

```
pascal ComponentResult DataHPreextend (DataHandler dh, long maxToAdd,
                    long *spaceAdded);
```

`dh` Identifies the calling program's connection to your data handler component.

`maxToAdd` Specifies the amount of space to add to the current data reference, in bytes. If the client program sets this parameter to 0, your component should add as much space as it can.

`spaceAdded` Contains a pointer to a long. Your component returns the number of bytes it was able to add to the data reference, in bytes.

DISCUSSION

This function is essentially analogous to the File Manager's `PBAllocContig` function. Your component should allocate contiguous free space. If there is not

sufficient contiguous free space to satisfy the request, your component should return a `dskFulErr` error code.

Client programs use this function in order to avoid incurring any space-allocation delay when capturing movie data.

Managing Data Handler Components

Your data handler component provides a number of functions that applications can use to manage their connections to your handler. The most important among these is `DataHTask`, which provides processor time to your handler. Applications should call this function often so that your handler has enough time to do its work.

Applications may call your handler's `DataHPlaybackHints` function in order to provide you with some guidelines about how those applications play to use the current data reference.

The `DataHFlushData` and `DataHFlushCache` functions allow applications to influence how your component manages its stored data.

For more information on managing data handlers, see "Managing the Data Handler" (page 16-13).

DataHTask

Client programs call your component's `DataHTask` function in order to cede processor time to your data handler.

```
pascal ComponentResult DataHTask (DataHandler dh);
```

`dh` Identifies the calling program's connection to your data handler component.

DISCUSSION

This function is essentially analogous to the Movie Toolbox's `MoviesTask` function. Client programs call this function in order to give your data handler component time to do its work. Your data handler uses this time to do its work. Because client programs will call this function frequently, and especially so

during movie playback or capture, your data handler should return control quickly to the client program.

DataHFlushCache

Your component discards the contents of any cached read buffers.

```
pascal ComponentResult DataHFlushCache (DataHandler dh);
```

`dh` Identifies the calling program's connection to your data handler component.

DISCUSSION

Client programs may call this function if they have, in some way, changed the container associated with the current data reference on their own. Under these circumstances, data your component may have read and cached in anticipation of future read requests from the client may be invalid.

Note that this function does not invalidate any queued read requests (made by calling your component's `DataHScheduleData` function).

DataHFlushData

Your component forces any data in its write buffers to be written to the device that contains the current data reference.

```
pascal ComponentResult DataHFlushData (DataHandler dh);
```

`dh` Identifies the calling program's connection to your data handler component.

DISCUSSION

This function is essentially analogous to the File Manager's `PBFlushFile` function. The client program may call this function after any write operation

(either DataHPutData or DataHWrite). Your component should do what is necessary to make sure that the data is written to the storage device that contains the current data reference.

DataHPlaybackHints

The DataHPlaybackHints function allows the client program to provide additional information to your component that you may use to optimize the operation of your data handler.

```
pascal ComponentResult DataHPlaybackHints (DataHandler dh, long flags,
                    unsigned long minFileOffset, unsigned long
                    maxFileOffset, long bytesPerSecond);
```

dh — Identifies the calling program's connection to your data handler component.

flags — Reserved for use by Apple Computer, Inc. Client programs should always set this parameter to 0.

minFileOffset — Together with the maxFileOffset parameter, specifies the range of data the client program anticipates using from the current data reference. This parameter specifies the earliest byte the program expects to use (that is, the minimum container offset value). If the client expects to access bytes from the beginning of the container, it should set this parameter to 0.

maxFileOffset — Specifies the latest byte the program expects to use (that is, the maximum container offset value). If the client expects to use bytes throughout the container, the client should set this parameter to –1.

bytesPerSecond — Indicates the rate at which your data handler must read data from the data reference in order to keep up with the client program's anticipated needs.

DISCUSSION

Your component should be prepared to have this function called more than once for a given data reference. For example, the Movie Toolbox calls this function whenever a movie's playback rate changes. This is a handy way for your data handler to track playback rate changes.

Completion Function

When client programs schedule asynchronous read or write operations (by calling your component's `DataHScheduleData` or `DataHWrite` functions), they furnish your component a data-handler completion function. Your component must call this function when it completes the read or write operation, whether the operation was a success or a failure.

Data handler Completion Function

The client program's completion function must present the following interface:

```
pascal void DHCompleteProc (Ptr request, long refcon, OSErr err);
```

`request` Specifies a pointer to the data that was associated with the read (`DataHScheduleData`) or write (`DataHWrite`) request. The client program uses this pointer to determine which request has completed.

`refcon` Contains a reference constant that the client program supplied to your data handler component when it made the original request.

`err` Indicates the success or failure of the operation. If the operation succeeded, set this parameter to 0. Otherwise, specify an appropriate error code.

Graphics Importer Components

Contents

About Graphics Importer Components

QuickTime 2.5 introduces a new way to draw still images. Graphics importer components provide a standard method for applications to open and display still images contained within graphics documents. Graphics importer components allow you to work with any type of image data, regardless of the file format or compression used in the document. You specify the document that contains the image, and the destination rectangle the image should be drawn into, and QuickTime handles the rest. More complex interactions are also supported.

The following example shows the basic functions you use to draw an image file.

```
void drawFile(const FSSpec *fss, const Rect *boundsRect)
{
    GraphicsImportComponent gi;
    GetGraphicsImporterForFile(fss, &gi);
    GraphicsImportSetBoundsRect(gi, boundsRect);
    GraphicsImportDraw(gi);
    CloseComponent(gi);
}
```

The same code can be used to display any image, regardless of the file format. QuickTime 2.5 supports the following image file formats: QuickDraw PICT, MacPaint, Photoshop (2.5 and 3.0), Silicon Graphics.rgb, GIF, and JFIF/JPEG. The new QuickTime image file format is also supported.

To obtain a graphics importer component for a particular file, use the Image Compression Manager's `GetGraphicsImporterForFile` function.

Note
If you expect to draw the same image more than once, you can improve performance by keeping the graphics importer component open, rather than creating and disposing it each time. ◆

QuickTime Image File Format

QuickTime's ability to include any compressed image data in a QuickDraw picture is a helpful feature from a compatibility perspective. However, it presents several technical challenges for applications that need to work with compressed image data contained within pictures. Determining if compressed data is present, and extracting it, requires special code installed in QuickDraw bottlenecks to detect and copy compressed data as it processes. Additional problems are posed by special cases such as multiple compressed images in a single file. The QuickTime image file (QTIF) format solves this and other issues.

QuickTime image files are intended to provide the most useful container for QuickTime compressed still images. The format uses the same atom-based structure as a QuickTime movie. (See chapter "MovieToolBox" for information on atoms.) There are two defined atom types: `'idsc'`, which contains an image description, and `'idat'`, which contains the image data. For a JPEG image, the image description atom contains a QuickTime image description describing the JPEG image's size, resolution, depth, and so on, and the image data atom contains the actual JPEG compressed data. (See chapter "xxxx" for additional information.) A QuickTime image file can also contain other atoms. For example, it can contain single-fork preview atoms. Because the QuickTime image file is a single fork format, it works well in cross-platform applications. On MacOS systems, QuickTime image files are identified by the file type `'qtif'`. Apple recommends using the filename extension `.QIF` to identify QuickTime image files on other platforms.

Graphics Importer Components Reference

Data Types

```
typedef ComponentInstance GraphicsImportComponent;

enum {
    GraphicsImporterComponentType = 'grip'
};
```

Functions

Specifying the Data Source

Graphics importer components use QuickTime data handler components to obtain their data. Applications, however, will use the graphics importer component functions described in this section, rather than directly calling a data handler. These functions allow the data source to be a file, a handle, or a QuickTime data reference.

GraphicsImportSetDataFile

Specifies the file that the graphics data resides in.

```
extern pascal ComponentResult GraphicsImportSetDataFile (
                    GraphicsImportComponent ci,
                    const FSSpec *theFile);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`theFile` A pointer to the file specification containing the graphics data.

DISCUSSION

The file will be opened for read access.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetDataFile

Returns the file that the graphics data resides in.

```
extern pascal ComponentResult GraphicsImportGetDataFile (
                    GraphicsImportComponent ci,
                    FSSpec *theFile);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`theFile` A pointer in which to return the file containing the graphics data.

DISCUSSION

You use this function to get the file system specification record for the file that the graphics data resides in. If the data source is not a file, the function returns `paramErr`.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportSetDataHandle

Specifies the handle that the graphics data resides in.

```
extern pascal ComponentResult GraphicsImportSetDataHandle (
                    GraphicsImportComponent ci,
                    Handle h);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`h` Specifies a handle containing graphics data.

DISCUSSION

The graphics imported component doesn't make a copy of this data. Therefore you must not dispose this handle until the graphics importer has been closed.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available

GraphicsImportGetDataHandle

Returns the handle that the graphics data resides in.

```
extern pascal ComponentResult GraphicsImportGetDataHandle (
                    GraphicsImportComponent ci,
                    Handle *h);
```

ci — Specifies the component instance that identifies your connection to the graphics importer component.

h — A pointer to a handle to return a handle containing the graphics data.

DISCUSSION

You use this function to get the handle that the graphics data resides in. If the data source is not a handle, the function returns paramErr.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available

GraphicsImportSetDataReference

Specifies the data reference that the graphics data resides in.

```
extern pascal ComponentResult GraphicsImportSetDataReference (
                    GraphicsImportComponent ci,
                    Handle dataRef,
                    OSType dataReType);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`dataRef`	A pointer to a handle to return a QuickTime data reference.
`dataReType`	A pointer to the data reference type.

DISCUSSION

Applications typically do not use this function. The `GraphicsImportSetDataFile` and `GraphicsImportSetDataHandle` functions both call this function, with the appropriate data reference and data reference type.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetDataReference

Returns the data reference that the graphics data resides in.

```
extern pascal ComponentResult GraphicsImportGetDataReference (
                    GraphicsImportComponent ci,
                    Handle *dataRef,
                    OSType *dataReType);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.

dataRef	A pointer to the handle to return a QuickTime data reference.
dataReType	A pointer to the data reference type.

DISCUSSION

You use this function to get the data reference that the graphics data resides in. Both the dataRef and dataReType parameters may be set to nil if the corresponding information is not desired. The GraphicsImportGetDataHandle and GraphicsImportGetDataFile functions call GraphicsImporGetDataReference and then manipulate the result accordingly.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available

GraphicsImportSetDataReferenceOffsetAndLimit

Specifies the data reference starting offset and data size limit.

```
extern pascal ComponentResult
                    GraphicsImportSetDataReferenceOffsetAndLimit (
                    GraphicsImportComponent ci,
                    unsigned long offset,
                    unsigned long limit);
```

ci	Specifies the component instance that identifies your connection to the graphics importer component.
offset	The byte offset of the image data from the beginning of the data reference.
limit	The data limit. This value is the maximum offset into the data reference that data may be read from.

DISCUSSION

A data reference typically refers to an entire file. However, there are times when the data being referenced is a part of a larger file. In these cases, it is necessary to indicate where the data begins in the data reference and where it ends. This function lets you specify the starting offset and ending offset. All requests to read data are then relative to the specified offset, and are pinned to the data size, so you cannot accidentally read off the end (or beginning) of the segment.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetDataReferenceOffsetAndLimit

Returns the data reference starting offset and data size limit.

```
extern pascal ComponentResult
                    GraphicsImportGetDataReferenceOffsetAndLimit (
                    GraphicsImportComponent ci,
                    unsigned long *offset,
                    unsigned long *limit);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`offset` A pointer to a value describing the byte offset of the image data from the beginning of the data reference.

`limit` A pointer to a value describing the data size limit.

DISCUSSION

This function returns the values set by the `GraphicsImportSetDataReferenceOffsetAndLimit` function. By default, `offset` is 0 and `limit` is `MaxInt` ($2^{32} - 1$).

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available

Validating and Retrieving Image Data

GraphicsImportValidate

Validates image data for a data reference.

```
pascal ComponentResult GraphicsImportValidate (
                    GraphicsImportComponent ci,
                    Boolean *valid)
```

ci	Specifies the component instance that identifies your connection to the graphics importer component.
valid	Pointer to a Boolean value. On return, this parameter is set to true if the the graphics importer component can draw the data reference. If the graphics importer component cannot draw the data reference, this parameter is set to false.

DISCUSSION

The GraphicsImportValidate functions allows a graphics importer component to determine if its current data reference contains valid image data. For example, a JFIF graphics importer component might check for the presence of a JFIF marker in the data reference. This function is provided for applications to use to determine what type of image data a particular file may contain. Sometimes a file may not have the correct file type or file extension. In this case, the application will not know which graphics importer component to use. By iterating through all graphics importer components and calling GraphicsImportValidate for each one, it may be possible to locate a graphics importer component that can draw the specified file.

Note
Not all graphics importer components implement this function. A component that does not implement the function will return the `badComponentSelector` result code. This does not indicate that the file is valid or invalid. ◆

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`badComponentSelector`	0x80008002	Component does not support the specified request code

GraphicsImportReadData

Reads image data.

```
extern pascal ComponentResult GraphicsImportReadData (
                    GraphicsImportComponent ci,
                    void *dataPtr,
                    unsigned long dataOffset,
                    unsigned long dataSize);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`dataPtr` A pointer to a memory block to receive the data.

`dataOffset` The offset of the image data within the data reference. The function begins reading image data from this offset.

`dataSize` The number of bytes of image data to read.

DISCUSSION

`GraphicsImportReadData` communicates with the appropriate data handler to retrieve image data. Typically only developers of graphics importer components will need to use this function. This function should always be used to retrieve data from the data source, rather than reading it directly.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available

Getting Image Characteristics

GraphicsImportGetNaturalBounds

Returns the bounding rectangle of an image.

```
extern pascal ComponentResult GraphicsImportGetNaturalBounds (
                    GraphicsImportComponent ci,
                    Rect *naturalBounds);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`naturalBounds`
A pointer to a rectangle structure describing the size of the bounding rectangle for the image.

DISCUSSION

You can use the `GraphicsImportGetNaturalBounds` function to determine the native size of the image associated with a graphics importer component. The natural bounds are always zero-based. This is a convenience function that simply calls `GraphicsImportGetImageDescription` and extracts the width and height fields.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available

GraphicsImportGetImageDescription

Returns image description information.

```
extern pascal ComponentResult GraphicsImportGetImageDescription (
                    GraphicsImportComponent ci,
                    ImageDescriptionHandle *desc);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`desc` A handle to an image description structure.

DISCUSSION

The `GraphicsImportGetImageDescription` function returns an image description structure containing information such as the format of the compressed data, its bit depth, natural bounds, and resolution. The caller is responsible for disposing of the returned image description handle.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetDataOffsetAndSize

Returns the offset and size of the compressed image data within a file.

```
extern pascal ComponentResult GraphicsImportGetDataOffsetAndSize (
                    GraphicsImportComponent ci,
                    unsigned long *offset,
                    unsigned long *size);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`offset`	A pointer to a value describing the byte offset of the image data from the beginning of the data source.
`size`	A pointer to a value describing the size of the image data in bytes.

DISCUSSION

This function returns the offset and size of the actual image data within the data source. By default the offset returned is 0 and the size returned is the size of the file. However, some graphics import components will override this function to skip over unneeded information at the beginning or end of the file.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

Setting Drawing Parameters

The functions described in this section allow you to specify various parameters for drawing operations, such as clipping, scaling, graphics mode, anddecompression quality. All of these functions are based on corresponding routines in the Image Compression Manager for working with image decompression sequences.

GraphicsImportSetBoundsRect

Defines the rectangle in which to draw an image.

```
extern pascal ComponentResult GraphicsImportSetBoundsRect (
                    GraphicsImportComponent ci,
                    const Rect *bounds);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.

bounds	A pointer to a rectangle structure describing the bounding rectangle into which the image will be drawn.

DISCUSSION

You use this function to define the rectangle into which the graphics image should be drawn. The function creates a transformation matrix to map the image's natural bounds to the specified bounds and then calls the GraphicsImportSetMatrix function.

Note
Because this function affects the transformation matrix, you should use the GraphicsImportSetMatrix function (page 17-17) instead of this function when you also need to specify more complex transformation of the matrix. ◆

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available

GraphicsImportGetBoundsRect

Returns the bounding rectangle for drawing.

```
extern pascal ComponentResult GraphicsImportGetBoundsRect (
                    GraphicsImportComponent ci,
                    Rect *bounds);
```

ci	Specifies the component instance that identifies your connection to the graphics importer component.
bounds	A pointer to a rectangle structure describing the bounding rectangle that has been defined for the image.

DISCUSSION

This is a convenience function that is implemented by calling `GraphicsImportGetMatrix` (page 17-18) and `GraphicsImportGetNaturalBounds` (page 17-13) and using the results to calculate the drawing rectangle.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportSetMatrix

Defines the transformation matrix to use for drawing an image.

```
extern pascal ComponentResult GraphicsImportSetMatrix (
                    GraphicsImportComponent ci,
                    const MatrixRecord *matrix);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`matrix` A pointer to a matrix structure that specifies how to transform the image during decompression. For example, you can use a transformation matrix to scale or rotate the image. To set the matrix to identity, pass `nil` in this parameter.

DISCUSSION

The `GraphicsImportSetMatrix` function establishes the transformation matrix to be applied to an image, which determines where and how it will be drawn.

Note
This function affects the bounding rectangle defined for the image. You can specify where an image will be drawn by setting either a transformation matrix or a bounding rectangle, but it is usually more convenient for applications to set a bounding rectangle using the `GraphicsImportSetBoundsRect` (page 17-15) function. ◆

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available

SEE ALSO

For more information about transformation matrices and the functions for working with them, see the Movie Toolbox chapter of *Inside Macintosh: QuickTime*.

GraphicsImportGetMatrix

Returns the transformation matrix to be used for drawing.

```
extern pascal ComponentResult GraphicsImportGetMatrix (
                    GraphicsImportComponent ci,
                    MatrixRecord *matrix);
```

ci	Specifies the component instance that identifies your connection to the graphics importer component.
matrix	A pointer to the transformation matrix that has been defined for the image.

DISCUSSION

The transformation matrix is initialized to the identity matrix when the graphics import component is instantiated.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available

GraphicsImportSetClip

Defines the clipping region for drawing.

```
extern pascal ComponentResult GraphicsImportSetClip (
                    GraphicsImportComponent ci,
                    RgnHandle clipRgn);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`clipRgn` A handle to a clipping region in the destination coordinate system. Set to `nil` to disable clipping. The graphics import component makes a copy of this region.

DISCUSSION

Because all drawing operations ignore the port clip, you must use this function to clip an image. The graphics importer component draws only that portion of the image that lies within the specified clipping region.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetClip

Returns the current clipping region.

```
extern pascal ComponentResult GraphicsImportGetClip (
                    GraphicsImportComponent ci,
                    RgnHandle *clipRgn);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

clipRgn	A handle to the clipping region that has been defined for the image. Returns nil if there is no clipping region.

DISCUSSION

The caller must dispose of the returned region handle.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available

GraphicsImportSetGraphicsMode

Sets the graphics transfer mode for an image.

```
extern pascal ComponentResult GraphicsImportSetGraphicsMode (
                    GraphicsImportComponent ci,
                    long graphicsMode,
                    const RGBColor *opColor);
```

ci	Specifies the component instance that identifies your connection to the graphics importer component.
graphicsMode	Specifies the graphics transfer mode to use for drawing the image. QuickTime supports the same graphics modes as Color QuickDraw's CopyBits function (described in *Inside Macintosh: Imaging with QuickDraw*) as well as any mode defined by the Image Compression manager, such as alpha modes.
opColor	A pointer to an RGB color structure describing the color to use for blending and transparent operations.

DISCUSSION

You can use this function to specify the graphics transfer mode and color to use for blending and transparent operations.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available

GraphicsImportGetGraphicsMode

Returns the graphics transfer mode for an image.

```
extern pascal ComponentResult GraphicsImportGetGraphicsMode (
                    GraphicsImportComponent ci,
                    long *graphicsMode,
                    RGBColor *opColor);
```

ci	Specifies the component instance that identifies your connection to the graphics importer component.
graphicsMode	A pointer to a long integer. The function returns the QuickDraw graphics transfer mode setting for the image.
opColor	A pointer to an RGB color structure. The function returns the color currently specified for blend and transparent operations.

DISCUSSION

Counldn't read comment.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available

GraphicsImportSetQuality

Sets the image quality value.

```
extern pascal ComponentResult GraphicsImportSetQuality (
                    GraphicsImportComponent ci,
                    CodecQ quality);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`quality` Specifies the desired image quality for decompression. Values for this parameter are on the same scale as compression quality. See page 3-57 of *Inside Macintosh: QuickTime* for a description of the defined quality constants.

DISCUSSION

The `quality` parameter controls how precisely the decompressor decompresses the image data. Some decompressors may choose to ignore some image data to improve decompression speed.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetQuality

Returns the image quality value.

```
extern pascal ComponentResult GraphicsImportGetQuality (
                    GraphicsImportComponent ci,
                    CodecQ *quality);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`quality`	A pointer to the currently specified quality value.

DISCUSSION

The `quality` value indicates how precisely the decompressor will decompress the image data. Some decompressors may choose to ignore some image data to improve decompression speed.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportSetSourceRect

Sets the source rectangle to use for an image.

```
extern pascal ComponentResult GraphicsImportSetSourceRect (
                    GraphicsImportComponent ci,
                    const Rect *sourceRect);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`sourceRect`	A pointer to a rectangle defining the portion of the image to decompress. This rectangle must lie within the boundary rectangle of the source image. Set to `nil` to use the entire image.

DISCUSSION

This function provides a way to use only a portion of the source image. Set the `sourceRect` parameter to `nil` to specify that the entire image source rectangle should be used.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available

GraphicsImportGetSourceRect

Returns the current source rectangle for an image.

```
extern pascal ComponentResult GraphicsImportGetSourceRect (
                    GraphicsImportComponent ci,
                    Rect *sourceRect);
```

ci	Specifies the component instance that identifies your connection to the graphics importer component.
sourceRect	A pointer to a rectangle structure. The function returns the source rectangle currently specified for the image.

DISCUSSION

This function returns the current source rectangle, as specified by the `GraphicsImportSetSourceRect` function. The default source rectangle is the image's natural bounds.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available

Drawing Images

GraphicsImportSetGWorld

Sets the graphics port and device for drawing.

```
extern pascal ComponentResult GraphicsImportSetGWorld (
                    GraphicsImportComponent ci,
                    CGrafPtr port,
                    GDHandle gd);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`port`	Specifies the destination graphics port or graphics world. Set to `nil` to use the current port.
`gd`	Specifies the destination graphics device. Set to `nil` to use the current device. If the `port` parameter specifies a graphics world, set this parameter to `nil` to use that graphics world's device.

DISCUSSION

The graphics world is initialized to the current port and device when the graphics importer component is opened. You can use this function to select another port or device.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetGWorld

Returns the current graphics port and device for drawing.

```
extern pascal ComponentResult GraphicsImportGetGWorld (
                    GraphicsImportComponent ci,
                    CGrafPtr *port,
                    GDHandle *gd);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`port` Returns the current destination graphics port. Set to `nil` if you are not interested in this information.

`gd` Returns the destination graphics device. Set to `nil` if you are not interested in this information.

DISCUSSION

This function returns the graphics port and device that will be used to draw the image. The graphics world is initialized to the current port and device when the graphics importer component is opened.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportDraw

Draws an image.

```
extern pascal ComponentResult GraphicsImportDraw
                    (GraphicsImportComponent ci);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

DISCUSSION

This function draws the image currently in use by the graphics import component to the graphics port and device specified by the `GraphicsImportSetGWorld` function. The `GraphicsImportDraw` function takes into account all settings you have specified for the image, such as the source rectangle, clipping region, graphics mode, and image quality.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

Saving Image Files

Graphics import components can save data in two formats: QuickDraw pictures and QuickTime image files. This capability is only needed by applications that perform file format translation. Applications that only wish to draw the image can use the `GraphicsImportDraw` function.

GraphicsImportSaveAsPicture

Creates a QuickDraw picture file.

```
extern pascal ComponentResult GraphicsImportSaveAsPicture (
                    GraphicsImportComponent ci,
                    const FSSpec *fss,
                    ScriptCode scriptTag);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`fss` A pointer to the file that is to receive the image.

`scriptTag` Specifies the script system in which the file name is to be displayed. If you have established the name and location of the file using one of the Standard File Package functions, use the script code returned in the reply record (`reply.sfScript`).

Otherwise, specify the system script by setting the `scriptTag` parameter to the value `smSystemScript`. See *Inside Macintosh: Files* for more information about script specification.

DISCUSSION

This function creates a new QuickDraw picture file containing the image currently in use by the graphics import component. If possible, the image will remain in the compressed format. For example, if the image is from a JFIF file, the picture will contain compressed JPEG data.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportSaveAsQuickTimeImageFile

Creates a QuickTime image file.

```
extern pascal ComponentResult GraphicsImportSaveAsQuickTimeImageFile (
                    GraphicsImportComponent ci,
                    const FSSpec *fss,
                    ScriptCode scriptTag);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`fss` A pointer to the file that is to receive the image.

`scriptTag` Specifies the script system in which the file name is to be displayed. If you have established the name and location of the file using one of the Standard File Package functions, use the script code returned in the reply record (`reply.sfScript`). Otherwise, specify the system script by setting the `scriptTag` parameter to the value `smSystemScript`. See *Inside Macintosh: Files* for more information about script specification.

DISCUSSION

This function creates a new QuickTime image file containing the image currently in use by the graphics import component. If possible, the image will remain in the compressed format. For example, if the image is from a JFIF file, the QuickTime image file will contain compressed JPEG data.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

QuickTime Settings Control Panel

Contents

This chapter discusses new features that have been added to the QuickTime Settings control panel.

New Features of the Control Panel

CD-ROM AutoStart

The CD-ROM AutoStart feature introduced with QuickTime 2.0 allows you to create CD-ROMs that automatically launch an application when the disc is inserted. This is useful for entertainment and educational titles because it makes it easier for users to begin using the software. QuickTime provides the low-level support necessary to recognize and launch AutoStart-enabled discs.

To create an AutoStart-enabled disc, you specify a document or application file as the AutoStart file. If the file you specified is an application, you may set it to invisible. Documents may not be invisible. If the AutoStart file is a document, QuickTime asks the Finder to launch the document. If an application is not available, the Finder will issue its normal warning, just as if the file had been double-clicked.

AutoStart works only with HFS discs. All information about AutoStart is contained in sector 0. The first two bytes in the sector must be either 0 or `'LK'`. The actual name of the AutoStart file is stored in the area allocated for the clipboard name. This area begins at offset 106. The first four bytes of the clipboard name field must contain the following value to indicate that the AutoStart file name follows: 0x006A 7068. After this 4-byte tag, 12 bytes of space remain, starting at offset 110. In these 12 bytes, the name of the AutoStart file is stored as a Pascal string, leaving up to 11 characters for the filename. The AutoStart file must reside in the root directory of the CD-ROM.

Because the AutoStart feature is based on the structure of an HFS disc, it is available only for the Macintosh. You can enable or disable AutoStart using the QuickTime Settings Control Panel.

AutoPlay for Audio CDs

QuickTime 2.5 allows you enable or disable the AutoPlay feature for audio CDs, and provides control over when you want a CD to play. If you want to use the Apple Audio CD Player or a similar application to control audio CD

playback, launch that application before inserting the CD. If the Apple Audio CD Player or a similar application is not running, the CD begins playing from track 1 automatically when you insert the disc. Otherwise, you control when to start and stop the audio using your application software.

QuickTime Music Synthesizer

Use this feature to select the default synthesizer for QuickTime to use.

QuickTime Music Architecture

Contents

This chapter describes QuickTime Music Architecture, which your applications and QuickTime movies can use to play sounds through a General MIDI synthesizer, through the Macintosh's built-in speaker, or through a hardware synthesizer. It also introduces atomic instruments. In previous versions of QuickTime Music Architecture, sounds were limited to the set of built-in sampled instruments. QuickTime 2.5 removes that limitation by introducing a public API and format for creating new atomic instruments and adding them to a QuickTime Music Architecture instrument component, increasing the number of available sounds.

In addition to other features, the QuickTime Music Architecture offers generalized access to synthesizers, which eliminates the requirements for an application to support a range of specific devices. It also has no limitation on the number of timbres (parts) available to an application or music track and offers a natural implementation of microtonal pitch scales, with 256 microtones in each semitone step.

Before reading about QuickTime Music Architecture, you should be familiar with QuickTime and QuickTime components. If you intend to create new instruments, you should be familiar with QT atoms.

This chapter first describes the basic capabilities of QuickTime Music Architecture and then provides a complete reference to its data structures, constants, functions, and result codes.

Note
The MoviePlayer and SimpleText applications allows you to open a standard MIDI file and convert it into a QuickTime music track. After the file is converted, the application prompts you to save the converted file as a QuickTime movie. Once saved, a movie controller is displayed and you can play the music. ◆

About QuickTime Music Architecture

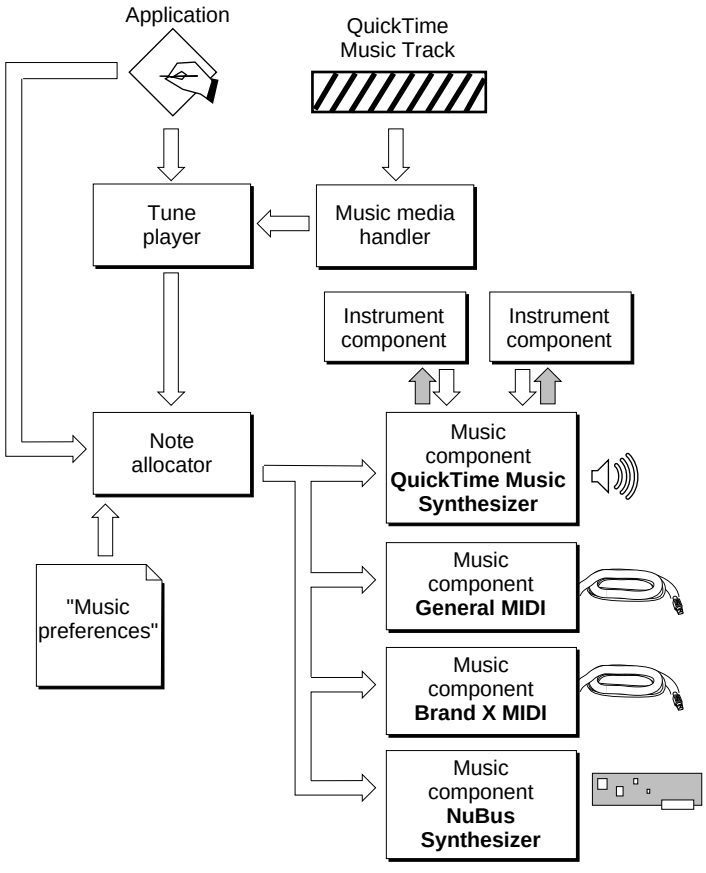
QuickTime Music Architecture (qtma) is composed of four software components—the note allocator, tune player, music component, and instrument component—and a set of music events.

This section describes the note allocator, tune player, and music components, and gives general guidelines for their use. In addition, this section introduces atomic instruments and the new instrument component, illustrates the format of the eight types of music events, and gives an overview of MIDI synthesizers.

QuickTime Music Architecture Components

You can use the QuickTime Music Architecture components to play individual notes, play tunes, control MIDI devices, and include atomic instruments to increase the number of sounds available. Figure 19-1 illustrates the relationships among the QuickTime Music Architecture components.

Figure 19-1 How QuickTime Music Architecture components work together



Note Allocator Component

You use the **note allocator component** to play individual notes. The calling application can specify which musical instrument sound to use and exactly which music synthesizer to play the notes on. The note allocator component also includes a default user interface utility that allows the user to choose instruments. The note allocator, unlike the tune player, provides no timing-related features to manage a sequence of notes. It's features are similar to a music component, although more generalized. Typically, an application

gains access to music components through the note allocator rather than directly.

To play a single note, an application must open an instance of the note allocator component and call `NANewNoteChannel` with a note request —typically to request a standard instrument within the General MIDI instrument set. A note channel is similar in some ways to a Sound Manager sound channel; it needs to be created and disposed, and can receive various commands. The note allocator provides an application-level interface for requesting note channels with particular attributes. The client specifies the desired polyphony and the desired tone. The note allocator returns a note channel that best satisfies the request.

With an open note channel, the application can call `NAPlayNote` while specifying the note's pitch and velocity. The note is played and remains playing until a second call to `NAPlayNote` is made specifying the same pitch, but with a velocity of zero. The velocity of zero causes the note to stop. The note allocator functions let you play individual notes, apply a controller change, apply a knob change, select an instrument based on a required tone, and modify or change the instrument type on an existing note channel.

There are calls for registering and unregistering a music component. During registration, the connections for that device are specified (typically, the connections are the MIDI Manager port and client IDs). There is also a call for querying the note allocator for registered devices, so that an application can offer a selection of the existing devices to the user. You can save configuration information in a preferences file.

Other note allocator functions offer a user interface and display copyright information.

Tune Player Component

The **tune player component** can accept entire sequences of musical notes and play them start to finish, asynchronously, with no further need for application intervention. It can also play portions of a sequence. An additional sequence or sequence section may be queued-up while one is currently being played. Queuing sequences provides a seamless way to transition between sections.

To use the tune player, an application need only open an instance of the tune player component, call `TuneSetHeader` with the appropriate header data, and call `TuneQueue` with the desired sequence data.

The tune player negotiates with the note allocator to determine which music component to use and allocates the necessary note channels. Any number of

sequences may be played simultaneously as long as there is sufficient polyphony (voices) within the music component. The tune player handles all aspects of timing, as defined by the sequence of music events. In addition, the tune player provides services to set the volume, and to stop and restart an active sequence.

Note
Using the QuickTime Movie Toolbox to play a movie that contains music data is often easier rather using the tune player directly. ◆

Music Components

Individual music components act as device drivers for each type of synthesizer attached to a particular computer. Two music components are provided with QuickTime 2.5: the QuickTime Music Synthesizer component, to play music out of the built-in speaker, and the General MIDI component, to play music on a General MIDI device attached to a serial port. To better understand the role of a music component, it helps to be familiar with QuickTime's model of a synthesizer. See "QuickTime Synthesizer Model" (page 19-27).

Applications do not usually call music components directly. Instead the note allocator or tune player handle music component interactions. Music components are mainly of interest to application developers who want to access low-level functionality of synthesizers and for developers of synthesizers (internal cards, MIDI devices, or software algorithms) who want to make the capabilities of their synthesizers available to QuickTime.

In order for an application to call a music component directly, you must first allocate a note channel and then use `NAGetNoteChannelInfo` and `NAGetRegisteredMusicDevice` to get the specific music component and part number.

You can use music component functions to obtain specific information about the current synthesizer, to find an instrument that best fits a requested type of sound, to play a note with a specified pitch, volume, and duration, and to get information about and work with MIDI entry points.

Other functions are for handling instruments and synthesizer parts. You can use these functions to initialize a part to a specified instrument and to get lists of available synthesizer instrument and drum kit names. You can also store modified instruments from a part into the modifiable instrument store, get

detailed information about each instrument from the synthesizer, and get information about and set knobs and controllers.

Instrument Components and Atomic Instruments

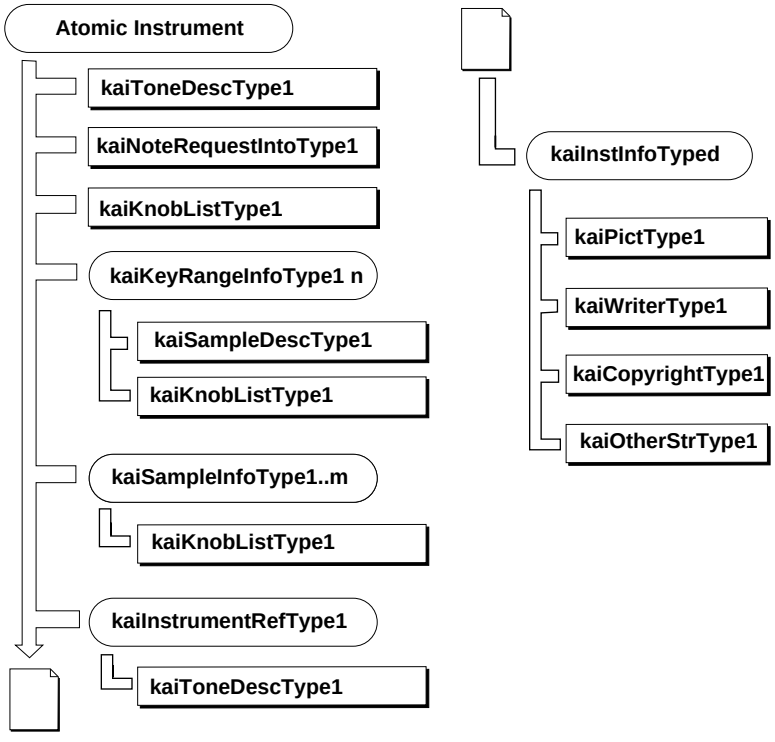
QuickTime 2.5 provides a public format for atomic instruments. These sounds may be embedded in a QuickTime movie, passed via a call to QuickTime, or dropped into the System Folder.

When initialized, the software synthesizer searches for components of type `'inst'`. These components may report a list of atomic instruments available to the software synthesizer. At present, `'inst'` components are only used by the QuickTime music synthesizer.

Atomic Instrument Format

The sounds are called **atomic instruments**, because you create them with QT atoms. (QT atoms are described in Chapter 1, "Movie Toolbox.") Using the QuickTime calls for manipulating atoms, you construct in memory a hierarchical tree of atoms with the data that describes the instrument (see Figure 19-2). The tree of atoms lives inside an atom container. There is one and only one root atom per container. Each atom has a four-character (32-bit) type, and a 32-bit ID. Each atom may be either an internal node, or a leaf atom with data.

Figure 19-2 An atomic instrument atom container.



You use the following types of atoms to construct an atomic instrument.

- Tone description atom (type `'tone'`). Every atomic instrument must have a tone description atom. The tone description atom contains the name of the instrument and a tone description structure. See "Tone Description Structure" on page 50.
- Knob list atom (type `'knbl'`). A knob list atom contains an instrument knob list. You can use knob list atoms to completely describe a custom instrument or to modify an instrument referred to by an instrument reference atom. See "Instrument Knob List" on page 47.
- Instrument reference atom (type `'iref'`). You can use an instrument reference atom to create a new atomic instrument based on an existing

instrument. An instrument reference atom contains a tone description that is to be modified by the knob list in the knob list atom.

- Note request atom (type `'ntrq'`). Note request atoms contain a note request information structure with information about the tone that's not included in the tone description. See "Note Request Information Structure" on page 58.
- Key range information atom (type `'sinf'`). To include your own sampled sounds in an atomic instrument, you must include one or more key range information atoms. Key range information atoms give instructions regarding which sample to play for a given range of pitches. For example, a piano may have a sample for the bass keys, one for the middle range, and one for the treble. Each key range information atom contains one or more sample description atoms. It may also include knob list atoms with knob settings for the sample (these settings override the instrument knob settings when the sample is played).
- Sample description atom (type `'sdsc'`). The sample description atom contains a sample description record with a description of the audio data. It refers via an ID number to the one or more sample data (`'sdat'`) atoms, which must also be present at the same level as the sample information atom. See "Atomic Instrument Sample Description Record" on page 47.
- Sample data atom (type `'sdat'`). The sample data atom contains the actual audio data. A sample data atom may be referred to by more than one sample description data.
- Instrument information type (type `'iinf'`). This optional atom contains atoms with information for the instrument About box.

Software Synthesizer Knobs

Knobs provide a way to modify the instrument sound, for example, by applying a tremolo. Atomic instruments for the software synthesizer are defined by some waveform data and a set of knob values. Typically, the instrument has a full list of knobs, and if the instrument contains more than a single sample, each sample contains values for several knobs that are tuned for that particular sample.

Knobs can be specified either by index or by ID. A nonzero value in the high byte of the 24-bit `number` field of an instrument knob record (page 19-46) or `knobID` field of a knob description record (page 19-51) indicates that it is an ID. The knob index ranges from 1 to the number of knobs; the ID is an arbitrary number. You should generally access knobs by ID, because knob IDs do not

change over different versions of the QuickTime software whereas knob index values might.

QuickTime Music Events

Music events specify the instruments and notes of a musical composition. A group of music events is called a **sequence**. A sequence of events may define a range of instruments and their characteristics and the notes and rests that, when interpreted, produce the musical composition.

The sequence of events required to produce music is usually contained in a QuickTime movie track, which uses a media handler to provide access to the tune player, or an application, which passes them directly to the tune player. QuickTime interprets and plays the music from the sequence data.

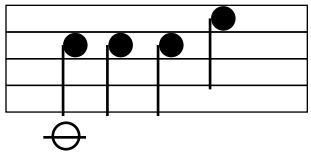
The events described in this section initialize and modify sound-producing music devices and define the notes and rests to be played.

Events are constructed as a group of long words. The uppermost four bits (nibble) of an event's long word defines its type.

First nibble	Long words	Event type
000x	1	Rest
001x	1	Note
010x	1	Controller
011x	1	Marker
1000	2	(reserved)
1001	2	Extended Note
1010	2	Extended Controller
1011	2	Knob
1100	2	(reserved)
1101	2	(reserved)
1110	2	(reserved)
1111	*n*	General

Durations of notes and rests are specified in units of the tune player's time scale (default 1/600 second). For example, consider the musical fragment shown in Figure 19-3.

Figure 19-3 A music fragment

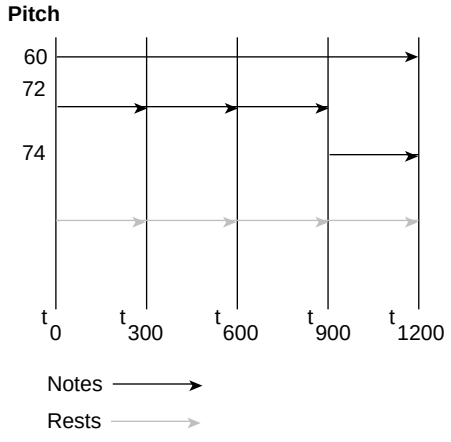


Assuming 120 beats-per-minute, and a tune player's scale of 600, each quarter note's duration is 300. Table 19-1 shows music track data for the music fragment in Figure 19-3. Figure 19-4 provides a graphical representation of that data.

Table 19-1 Music track data

Track data	What it does
NOTE Part 0, pitch 60, duration 1200	plays for four beats
NOTE Part 0, pitch 72, duration 300	plays for one beat
REST duration 300	delays start of next note
NOTE Part 0, pitch 72, duration 300	plays for one beat
REST duration 300	delays start of next note
NOTE Part 0, pitch 72, duration 300	plays for one beat
REST duration 300	delays start of next note
NOTE Part 0, pitch 74, duration 300	plays for one beat
REST duration 300	delays start of next note

Figure 19-4 Duration of notes and rests



The General event specifies the types of instruments or sounds used for the subsequent Note events. The Note event causes a specific instrument, previously defined by a General event, to play a note at a particular pitch and velocity for a specified duration of time.

Additional event types allow sequences to apply controller effects to instruments, define rests, and modify instrument knob values. The entire sequence is closed with a Marker event.

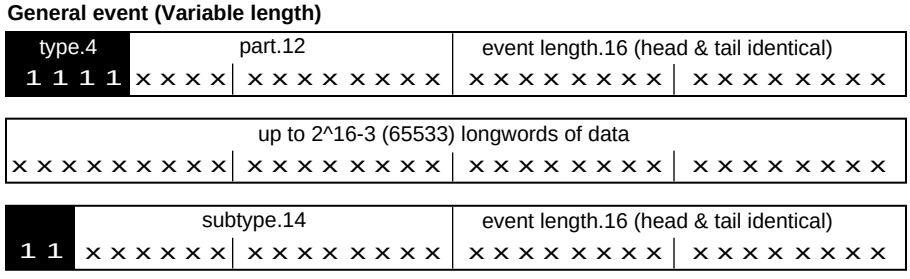
In most cases, the standard Note and Controller events (two long words) are sufficient for an application's requirements. The Extended Note event provides wider pitch range and fractional pitch values. The Extended Controller event expands the number of instruments and controller values over that allowed by a Controller event.

The following sections describe the event types in detail.

General Event

You use the General event to inform QuickTime Music Architecture of a synthesizer to be used by subsequent events. The tune player call, `TuneSetHeader`, receives the General event shown in Figure 19-5.

Figure 19-5 A note request General event



General event type	first nibble value = `1111`
Part number	unique part identifier
Event length	head: number of words in event
Data words	depends on subtype
Subtype	8-bit unsigned subtype
Event length	tail: must be identical to head
Event tail	first nibble of last word = `11XX`

The part number bit field contains a unique identifier that is later used to match Note, Knob, and Controller events to a specific part. For example, to play a note the application uses the part number to specify which instrument will play the note. The General event allows part numbers of up to 12 bits. The standard Note and Controller events allow part numbers of up to 5 bits; the Extended Note and Extended Controller events allow 12-bit part numbers.

The event length bit fields contained in the first and last words of the message are identical and are used as a message format check and to move back and forth through the message.

The data words field is a variable length field containing information unique to the subtype of the General event. The subtype bit field indicates the subtype of General event. There are nine subtypes:

- A note request General event (`kGeneralEventNoteRequest`) has a subtype of 1. It encapsulates the note request data structure used to define the instrument or part.

- A part key General event (`KGeneralEventPartKey`) has a subtype of 4. It sets a pitch offset for the entire part so that every subsequent note played on that part will be altered in pitch by the specified amount.
- A tune difference General event (`kGeneralEventTuneDifference`) has a subtype of 5. It contains a standard sequence, with end marker, for the tune difference of a sequence piece. Using a tune difference event is similar to using key frames with compressed video sequences. (This subtype halts QuickTime 2.0 music).
- An atomic instrument General event (`kGeneralEventAtomicInstrument`) has a subtype of 6. It encapsulates an atomic instrument.
- A knob General event (`kGeneralEventKnob`) has a subtype of 7. It contains knob ID/knob value pairs. The smallest event is four long words.
- A MIDI channel General event (`kGeneralEventMIDIChannel`) has a subtype of 8. It is used in a tune header. One long word identifies the MIDI channel it originally came from.
- A part change General event (`kGeneralEventPartChange`) has a subtype of 9. It is used in a tune sequence where one long word identifies the tune part that can now take over the part's note channel. (This subtype halts QuickTime 2.0 music.)
- A no-op General event (`kGeneralEventNoOp`) has a subtype of 10. It does nothing in QuickTime 2.5 but it halts QuickTime 2.0 music.
- A notes-used General event (`kGeneralEventUsedNotes`) has a subtype of 11. It is four long words specifying which MIDI notes are actually used.

Macro calls are used to stuff the General event's head and tail long words, but not the structures described above:

```
_StuffGeneralEvent(w1, w2, instrument, subType, length)
```

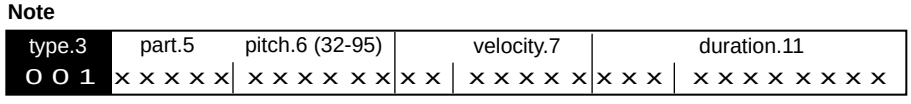
Macros are used to extract field values from the event's head and tail long words.

```
qtma_XInstrument(m, l)
qtma_GeneralSubtype(m, l)
qtma_GeneralLength(m, l)
```

Note Event and Extended Note Event

The standard Note event (Figure 19-6) supports most music requirements. The Note event allows up to 32 instruments and supports the traditional equal tempered scale. The Extended Note event (Figure 19-7) provides a wider range of pitch values, microtonal values to define any pitch, and extended note duration. The Extended Note event requires two long words; the standard Note event requires only one.

Figure 19-6 Note event



Note event type	first nibble value = `001X`
Part number	unique part identifier
Pitch	numeric value of 0–63, mapped to 32–95
Velocity	0–127, 0 = no audible response (but used to indicate a NOTE OFF)
Duration	specifies how long to play the note in units defined by the media time scale or tune player time scale

The part number field contains the unique part identifier initially used during the `TuneSetHeader` call.

The pitch bit field allows a range from 0-63, which is mapped to the values 32–95 representing the traditional equal tempered scale. For example, the value 28 (mapped to 60) is middle C.

The velocity bit field allows a range from 0–127. A velocity value of 0 produces silence.

The duration bit field defines the number of units of time during which the part will play the note. The units of time are defined by the media time scale or tune player time scale.

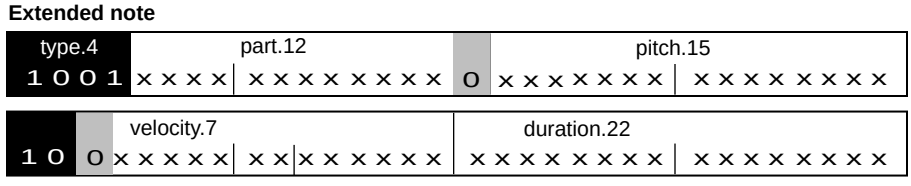
Macro call used to stuff the Note event's long word:

```
_StuffNoteEvent(x, instrument, pitch, volume, duration)
```

Macro calls used to extract fields from the Note event's long word:

```
qtma_Instrument(x)
qtma_NotePitch(x)
qtma_NoteVelocity(x)
qtma_NoteDuration(x)
```

Figure 19-7 Extended Note event



Extended Note event type	first nibble value = `1001`
Part number	unique part identifier
Pitch	0–127 standard pitch, 60 = middle C `0x01.00` … `0x7F.00` allowing 256 microtonal divisions between each notes in the traditional equal tempered scale
Duration	specifies how long to play the note in units defined by media time scale or tune player time scale
Velocity	0–127 where 0 = no audible response (but used to indicate a NOTE OFF)
Event tail	first nibble of last word = `10XX`

The part number bit field contains the unique part identifier initially used during the `TuneSetHeader` call.

If the pitch field is less than 128, it is interpreted as an integer pitch where 60 is middle C. If the pitch is 128 or greater, it is treated as a fixed pitch.

Microtonal pitch values are produced when the 15 bits of the pitch field are split. The upper 7 bits defines the standard equal tempered note and the lower 8 bits defines 256 microtonal divisions between the standard notes.

Macro call used to stuff the extended Note event's long words:

```
_StuffXNoteEvent(w1, w2, instrument, pitch, volume, duration)
```

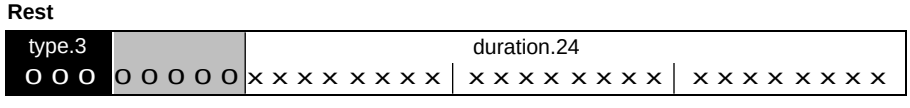
Macro calls used to extract fields from the extended Note event's long words:

```
qtma_XInstrument(m, l)
qtma_XNotePitch(m, l)
qtma_XNoteVelocity(m, 1)
qtma_XNoteVolume(m, l)
qtma_XNoteDuration(m, l)
```

Rest Event

The Rest event (Figure 19-8) specifies the period of time, defined by either the media time scale or the tune player time scale, until the next Note event in the sequence will be played.

Figure 19-8 Rest event



Rest event type	first nibble value = `000X`
Duration	specifies the number of units of time until the next Note event is played in units defined by media time scale or tune player time scale

Macro call used to stuff the Rest event's long word:

```
_StuffRestEvent(x, duration)
```

Macro call used to extract the Rest event's duration value:

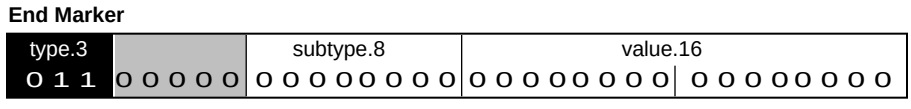
```
qtma_RestDuration(x)
```

Note
Rest events are not used to cause silence in a sequence, but to define the start of subsequent Note events. ◆

Marker Event

The Marker event has three subtypes. The end Marker event (Figure 19-9) marks the end of a series of events. The beat Marker event marks the beat and the tempo Marker event indicates the tempo.

Figure 19-9 Marker event of subtype End



Marker event type	first nibble value = `011X`
Subtype	8-bit unsigned subtype
Value	16-bit signed value

The Marker subtype bit field contains zero for an end marker (`kMarkerEventEnd`), 1 for a beat marker (`kMarkerEventBeat`), or 2 for a tempo marker (`kMarkerEventTempo`).

The value bit field varies according to the subtype:

- For an end Marker event, a value of 0 means stop; any other value is reserved.
- For a beat Marker event, a value of 0 is a single beat (a quarter note); any other value indicates the number of fractions of a beat in 1/65536 beat.
- For a tempo Marker event, the value is the same as a beat marker, but indicates that a tempo event should be computed (based on where the next beat or tempo marker is) and emitted upon export.

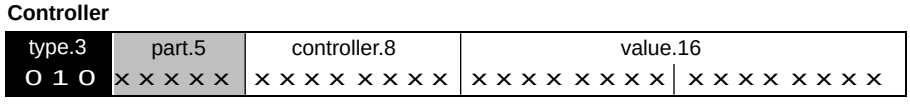
Macro calls used to extract fields from the Marker events long word:

```
qtma_MarkerSubtype(x)
qtma_MarkerValue(x)
```

Controller Event and Extended Controller Event

The Controller event (Figure 19-10) changes the value of a controller on a specified part. The Extended Controller event (Figure 19-11) allows parts and controllers beyond the range of the standard controller event.

Figure 19-10 Controller event



Controller event type	first nibble value = `010X`
Part	unique part identifier
Controller	controller to be applied to instrument
Value	8.8 bit fixed point signed controller specific value

For a list of currently supported controller types see "Controller Numbers" (page 19-33).

The part field contains the unique part identifier initially used during the `TuneSetHeader` call.

The controller bit field is a value that describes the type of controller used by the part.

The value bit field is specific to the selected controller.

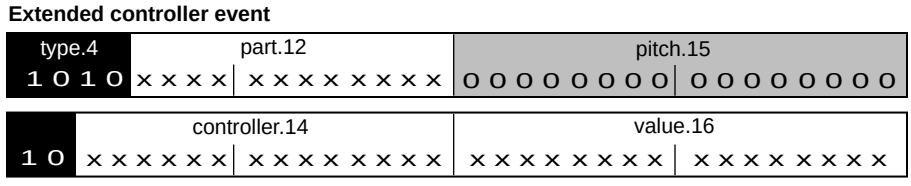
Macro call used to stuff the controller event's long word:

```
_StuffControlEvent(x, instrument, control, value)
```

Macro calls used to extract fields from the controller event's long word:

```
qtma_Instrument(x)
qtma_ControlController(x)
qtma_ControlValue(x)
```

Figure 19-11 Extended Controller event



Extended Controller type	first nibble value = `1010`
Part	instrument index for controller
Controller	controller for instrument
Value	signed controller specific value
Event tail	first nibble of last word = `10XX`

The part field contains the unique part identifier initially used during the `TuneSetHeader` call.

The controller bit field contains a value that describes the type of controller to be used by the part.

The value bit field is specific to the selected controller.

Macro call used to stuff the Extended Controller event's long words:

```
_StuffXControlEvent(w1, w2, instrument, control, value)
```

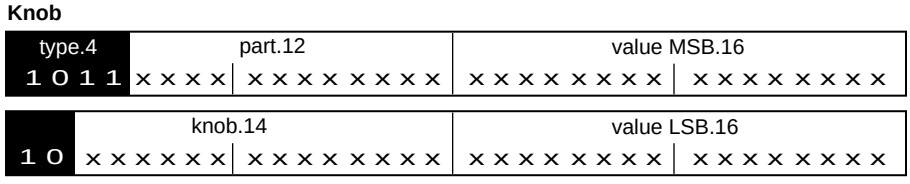
Macro calls used to extract fields from the Extended Controller event's long words:

```
qtma_XInstrument(m, l)
qtma_XControlController(m, l)
qtma_XControlValue(m, l)
```

Knob Event

The Knob event is used to modify a particular knob within a specified part.

Figure 19-12 Knob event



Knob event type	first nibble value = `1011`
Part	unique part identifier
Knob number	knob number within specified part
Knob value (LSW (0–15))	lower 16 bits of knob value
Knob value (MSW (16–31))	upper 16 bits of knob value
Event tail	first nibble of last word = `10XX`

The part field contains the unique part identifier initially used during the `TuneSetHeader` call.

The knob number bit field identifies the knob to be changed.

The 32-bit value composed of the lower 16- and upper 16-bit field values is used to alter the specified knob.

Macro call used to stuff the Knob event's long words:

```
_StuffKnobEvent(w1, w2, instrument, knob, value)
```

Macro calls used to extract fields from the Knob event's long words:

```
qtma_XInstrument(m, l)
qtma_KnobValue(m, l)
```

QuickTime Synthesizer Model

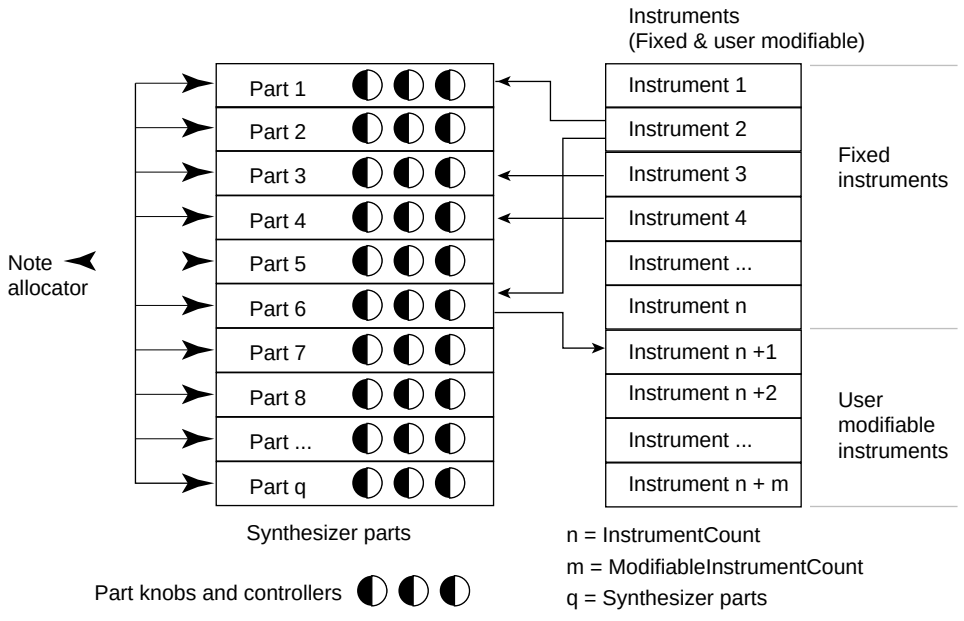
Although many kinds of synthesizers have been built, QuickTime Music Architecture uses one specific model. The QuickTime Music Architecture music synthesizer model is described in this section.

A synthesizer contains a number of parts and instruments. An instrument is a very specific description of the type of sound produced. Parts can be thought of as slots in which the user installs particular instruments.

An instrument is accessible only after it is loaded into one of the synthesizer's parts. A synthesizer has parts for fixed and user-modifiable instruments. An instrument loaded into a part can be modified by changing the value of one of its knobs and saving it in one of the modifiable instrument slots under a new name.

The diagram below illustrates a typical synthesizer. The illustration shows the total number of parts available from the synthesizer, a group of fixed instruments, 1 through n, and a group of user modifiable instruments, n+1 through n+m.

Figure 19-13 Typical synthesizer



In the illustration above, part 6 contains a user-modifiable instrument. It uses the same instrument as part 1. One instrument can be used by two separate parts. After an instrument is loaded into multiple parts, either part can be modified through its knobs. Knobs allow you to produce a unique variation from the original instrument. Knobs apply to the entire synthesizer and not to a particular instrument. Knobs typically control audio effects such as reverb that are built into a synthesizer.

In the illustration, part 6 is saved to the user modifiable instrument n+1. Modified parts cannot be saved to a fixed instrument slot.

QuickTime Music Architecture Reference

This section describes the constants, data structures, functions, and result codes provided by QuickTime Music Architecture.

Constants

This section describes the constants provided by QuickTime Music Architecture.

Atom Types for Atomic Instruments

These constants specify the types of atoms used to build atomic instruments. Atomic instruments are described in "Instrument Components and Atomic Instruments" (page 19-12).

```
enum {
    kaiToneDescType                 = 'tone',
    kaiNoteRequestInfoType          = 'ntrq',
    kaiKnobListType                 = 'knbl',
    kaiKeyRangeInfoType             = 'sinf',
    kaiSampleDescType               = 'sdsc',
    kaiSampleDataType               = 'sdat',
    kaiInstRefType                  = 'iref',
    kaiInstInfoType                 = 'iinf',
    kaiPictType                     = 'pict',
    kaiWriterType                   = '©wrt',
    kaiCopyrightType                = '©cpy',
    kaiOtherStrType                 = 'str '
};
```

Constant descriptions

`kaiToneDescType` A tone atom, which describes the tone. It contains a tone description structure (page 19-50).

`kaiNoteRequestInfoType` A note request information atom, which contains a note request information structure (page 19-58). The note request information structure includes information about a tone that is not in the tone description. Use a note request information atom when embedding an instrument in a sample description of a QuickTime movie. If this atom is absent, QuickTime assumes "reasonable" values for polyphony.

`kaiKnobListType` A knob list atom, which specifies values for one or more knobs. It contains an instrument knob list structure

(page 19-47). Use it with a custom instrument, a modified built-in instrument, or as part of a sample.

`kaiKeyRangeInfoType`
: Use a key range information atom to include a sampled sound in an atomic instrument. A key range information atom contains several other atoms. It also refers, via an ID, to one or more sibling sample info (`kaiSampleInfoType`) atoms.

`kaiSampleDescType`
: A sample description atom, which contains an atomic instrument sample description record (page 19-47).

`kaiSampleDataType`
: A sample data atom, which contains the actual audio data.

`kaiInstRefType`
: An instrument reference atom, which contains a tone description to be modified by a knob list atom.

`kaiInstInfoType`
: An instrument information atom, which contains four optional atoms with information for the instrument About box.

`kaiPictType`
: A picture atom that includes the graphic used in the instrument About box.

`kaiWriterType`
: A text atom that has the author information used in instrument the About box.

`kaiCopyrightType`
: A text atom that has the copyright information used in the instrument About box.

`kaiOtherStrType`
: A text atom that has additional information for the instrument About box.

`kaiSampleInfoType`
: A text atom that contains a sample data (`kiaSampleDataType`) atom.

Instrument Knob Flags

These flags are used in the `flags` field of an instrument knob list structure (page 19-47) to indicate what to do if a requested knob is not in the list.

```
enum {
    kInstKnobMissingUnknown         = 0,
    kInstKnobMissingDefault         = 1 << 0
};
```

Constant descriptions

`kInstKnobMissingUnknown`
If the requested knob is not in the list, do not set its value.

`kInstKnobMissingDefault`
If the requested knob is not in the list, use its default value.

Loop Type Constants

You can use these constants in the `loopType` field of an atomic instrument sample description record (page 19-47) to indicate the type of loop you want.

```
enum {
    kMusicLoopTypeNormal            = 0,
    kMusicLoopTypePalindrome        = 1
};
```

Constant descriptions

`loopTypeNormal` Use a regular loop.

`loopTypeAlternating`
Take the wave form and reverse its sin waves and its timing. This produces a wave form with odd harmonics.

Music Component Type

Use this constant to specify a QuickTime music component.

```
enum {
    kMusicComponentType     = 'musi'
};
```

Constant description

`kMusicComponentType`
The QuickTime Music Architecture music component type.

Synthesizer Type Constants

You can use these constants in a tone description structure (page 19-50) to specify the type of synthesizer you want to produce the tone.

```
enum {
    kSoftSynthComponentSubType      = 'ss  ',
    kGMSynthComponentSubType        = 'gm  '
};
```

Constant descriptions

`kSoftSynthComponentSubType`
: Use the QuickTime music synthesizer. This is the built-in synthesizer

`kGMSynthComponentSubType`
: Use the General MIDI synthesizer.

Synthesizer Description Flags

These flags describe various characteristics of a synthesizer. They are used in the `flags` field of the synthesizer description structure (page 19-48).

```
enum {
    kSynthesizerDynamicVoice        = 1,
    kSynthesizerUsesMIDIPort        = 2,
    kSynthesizerMicrotone           = 4,
    kSynthesizerHasSamples          = 8,
    kSynthesizerMixedDrums          = 6,
    kSynthesizerSoftware            = 32,
    kSynthesizerHardware            = 64,
    kSynthesizerDynamicChannel      = 128,
    kSynthesizerHogsSystemChannel   = 256,
    kSynthesizerSlowSetPart         = 1024,
    kSynthesizerOffline             = 4096,
    kSynthesizerGM                  = 16384
};
```

Constant descriptions

`kSynthesizerDynamicVoice`
: Voices can be assigned to parts on the fly with this synthesizer (otherwise, polyphony is very important).

`kSynthesizerUsesMIDIPort`
: This synthesizer must be patched through a MIDI system, such as the MIDI Manager or OMS.

`kSynthesizerMicrotone`
: This synthesizer can play microtonal scales.

`kSynthesizerHasSamples`
: This synthesizer has some use for sampled audio data.

`kSynthesizerMixedDrums`
: Any part of this synthesizer can play drum parts.

`kSynthesizerSoftware`
: This synthesizer is implemented in main CPU software and uses CPU cycles.

`kSynthesizerHardware`
: This synthesizer is a hardware device, not a software synthesizer or MIDI device.

`kSynthesizerDynamicChannel`
: This synthesizer can move any part to any channel or disable each part. For MIDI devices only.

`kSynthesizerHogsSystemChannel`
: Even if the `kSynthesizerDynamicChannel` bit is set, this synthesizer always responds on its system channel. For MIDI devices only.

`kSynthesizerSlowSetPart`
: This synthesizer does not respond rapidly to the various set part and set part instrument calls.

`kSynthesizerOffline`
: This synthesizer can enter an off-line synthesis mode.

`kSynthesizerGM`
: This synthesizer is a General MIDI device.

Controller Numbers

The controller numbers used by QuickTime are mostly identical to the standard MIDI controller numbers. These are signed 8.8 values. The full range, therefore, is `-128.00` to `127+127/128 (or 0x8000 to 7FFF)`.

All controls default to zero except for volume and pan.

Pitch bend is specified in fractional semitones, which eliminates the restrictions of a pitch bend range. You can bend as far as you want, any time you want.

The last 16 controllers (113–128) are global controllers. Global controllers respond when the part number is given as 0, indicating the entire synthesizer.

```
enum {
    kControllerModulationWheel          = 1,
    kControllerBreath                   = 2,
    kControllerFoot                     = 4,
    kControllerPortamentoTime           = 5,
    kControllerVolume                   = 7,
    kControllerBalance                  = 8,
    kControllerPan                      = 10,
    kControllerExpression               = 11,
    kControllerLever1                   = 16,
    kControllerLever2                   = 17,
    kControllerLever3                   = 18,
    kControllerLever4                   = 19,
    kControllerLever5                   = 80,
    kControllerLever6                   = 81,
    kControllerLever7                   = 82,
    kControllerLever8                   = 83,
    kControllerPitchBend                = 32,
    kControllerAfterTouch               = 33,
    kControllerSustain                  = 64,
    kControllerSostenuto                = 66,
    kControllerSoftPedal                = 67,
    kControllerReverb                   = 91,
    kControllerTremolo                  = 92,
    kControllerChorus                   = 93,
    kControllerCeleste                  = 94,
    kControllerPhaser                   = 95,
    kControllerEditPart                 = 113,
    kControllerMasterTune               = 114
};
```

Constant descriptions

`kControllerModulationWheel`
: This controller controls the modulation wheel. A modulation wheel adds a warble.

`kControllerBreath`
: This controller controls breath.

`kControllerFoot`
: This controller controls the foot pedal.

`kControllerPortamentoTime`
: This controller adjusts the slur between notes. Set the time

to 0 to turn off portamento; there is no separate control to turn portamento on and off.

`kControllerVolume` This controller controls volume.

`kControllerBalance` This controller controls balance between channels.

`kControllerPan` This controller controls balance on the QuickTime music synthesizer and some others. Values are 256– 512, corresponding to left to right.

`kControllerExpression`
This controller provides a second volume control.

`kControllerLever1` through `kControllerLever8`
These are all general purpose controllers.

`kControllerPitchBend`
This controller bends the pitch. Pitch bend is specified in positive and negative semitones, with 7 bits per fraction.

`kControllerAfterTouch`
This controller controls channel pressure.

`kControllerSustain` This controller controls the sustain effect. The value is a Boolean—positive for on, 0 or negative for off.

`kControllerSostenuto`
This controller controls sostenuto.

`kControllerSoftPedal`
This controller controls the soft pedal.

`kControllerReverb` This controller controls reverb.

`kControllerTremolo` This controller controls tremolo.

`kControllerChorus` This controller controls the amount of signal to feed to the chorus special effect unit.

`kControllerCeleste` This controller controls the amount of signal to feed to the celeste special effect unit.

`kControllerPhaser` This controller controls the amount of signal to feed to the phaser special effect unit.

`kControllerEditPart`
This controller sets the part number for which editing is occurring. For synthesizers that can edit only one part.

`kControllerMasterTune`
This controller offsets the entire synthesizer in pitch.

Controller Range

These constants specify the maximum and minimum values for controllers.

```
enum {
    kControllerMaximum      = 0x7FFF,
    kControllerMinimum      = 0x8000
};
```

Constant descriptions

`kControllerMaximum`
The maximum value a controller can be set to.

`kControllerMinimum`
The minimum value a controller can be set to.

Drum Kit Numbers

These constants specify the first and last drum kit numbers available to General MIDI drum kits.

```
enum {
    kFirstDrumkit   = 16384,
    kLastDrumkit    = (kFirstDrumkit + 128)
};
```

Constant description

`kFirstDrumkit` The first number in the range of drum kit numbers, which corresponds to "no drum kit." The standard drum kit is `kFirstDrumKit+1=16385`.

`kLastDrumkit` The last number in the range of drum kit numbers.

Tone Fit Flags

These flags are returned by the `MusicFindTone` function (page 19-104) to indicate how well an instrument matches the tone description.

```
enum {
    kInstrumentMatchSynthesizerType     = 1,
    kInstrumentMatchSynthesizerName     = 2,
```

```
    kInstrumentMatchName                = 4,
    kInstrumentMatchNumber              = 8,
    kInstrumentMatchGMNumber            = 16
};
```

Constant descriptions

`kInstrumentMatchSynthesizerType`
: The requested synthesizer type was found.

`kInstrumentMatchSynthesizerName`
: The particular instance of the synthesizer requested was found.

`kInstrumentMatchName`
: The instrument name in the tone description matched an appropriate instrument on the synthesizer.

`kInstrumentMatchNumber`
: The instrument number in the tone description matched an appropriate instrument on the synthesizer.

`kInstrumentMatchGMNumber`
: The General MIDI equivalent was used to find an appropriate instrument on the synthesizer.

Knob Flags

Knob flags specify characteristics of a knob. They are used in the `flags` field of a knob description record. Some flags describe the type of values a knob takes and others describe the user interface. Knob type flags are mutually exclusive, so only one should be set (all knob type flag constants begin "`kKnobType`").

```
enum {
    kKnobReadOnly               = 16,
    kKnobInterruptUnsafe        = 32,
    kKnobKeyrangeOverride       = 64,
    kKnobGroupStart             = 128,
    kKnobFixedPoint8            = 1024,
    kKnobFixedPoint16           = 2048,
    kKnobTypeNumber             = 0 << 12,
    kKnobTypeGroupName          = 1 << 12,
    kKnobTypeBoolean            = 2 << 12,
    kKnobTypeNote               = 3 << 12,
```

```
    kKnobTypePan                = 4 << 12,
    kKnobTypeInstrument         = 5 << 12,
    kKnobTypeSetting            = 6 << 12,
    kKnobTypeMilliseconds       = 7 << 12,
    kKnobTypePercentage         = 8 << 12,
    kKnobTypeHertz              = 9 << 12,
    kKnobTypeButton             = 10 << 12
};
```

Constant descriptions

`kKnobReadOnly` The knob value cannot be changed by the user or with a set knob call.

`kKnobInterruptUnsafe` Alter this knob only from foreground task time.

`kKnobKeyrangeOverride` The knob can be overridden within a single key range (software synthesizer only).

`kKnobGroupStart` The knob is first in some logical group of knobs.

`kKnobFixedPoint8` Interpret knob numbers as fixed-point 8-bit.

`kKnobFixedPoint16` Interpret knob numbers as fixed-point 16-bit.

`kKnobTypeNumber` The knob value is a numerical value.

`kKnobTypeGroupName` The name of the knob is really a group name for display purposes.

`kKnobTypeBoolean` The knob is an on/off knob. If the range of the knob (as specified by the low value and high value in the knob description record) is greater than one, the knob is a multi-checkbox field.

`kKnobTypeNote` The knob value range is equivalent to MIDI keys.

`kKnobTypePan` T knob value is the pan setting and is within a range (as specified by the low value and high value in the knob description record) that goes from left to right.

`kKnobTypeInstrument` The knob value is a reference to another instrument number.

`kKnobTypeSetting` The knob value is one of *n* different discrete settings—for example, items on a pop-up menu.

`kKnobTypeMilliseconds`
The knob value is in milliseconds.

`kKnobTypePercentage`
The knob value is a percentage of the range.

`kKnobTypeHertz` The knob value represents frequency.

`kKnobTypeButton` The knob is a momentary trigger push button.

Knob Value Constants

These constants specify unknown or default knob values and are used in various get knob and set knob calls.

```
enum {
    kUnknownKnobValue       = 0x7FFFFFFF,
    kDefaultKnobValue       = 0x7FFFFFFE
};
```

Constant descriptions

`kUnknownKnobValue` Couldn't find the specified knob value.

`kDefaultKnobValue` Set this knob to its default value.

Music Packet Status

These constants are used in the `reserved` field of the MIDI packet structure (page 19-52).

```
enum {
    kMusicPacketPortLost        = 1,
    kMusicPacketPortFound       = 2,
    kMusicPacketTimeGap         = 3
};
```

Constant descriptions

`kMusicPacketPortLost`
The application has lost the default input port.

`kMusicPacketPortFound`
The application has retrieved the input port from the previous owner.

`kMusicPacketTimeGap`
: The last byte of the packet specifies how long to keep the MIDI line silent in milliseconds, after sending the packet.

Atomic Instrument Information Flags

These constants specify what pieces of information about an atomic instrument the caller is interested in and are passed to the `MusicGetPartAtomicInstrument` function.

```
enum {
    kGetAtomicInstNoExpandedSamples = 1 << 0,
    kGetAtomicInstNoOriginalSamples = 1 << 1,
    kGetAtomicInstNoSamples         = kGetAtomicInstNoExpandedSamples |
        kGetAtomicInstNoOriginalSamples,
    kGetAtomicInstNoKnobList        = 1 << 2,
    kGetAtomicInstNoInstrumentInfo  = 1 << 3,
    kGetAtomicInstOriginalKnobList  = 1 << 4,
    kGetAtomicInstAllKnobs          = 1 << 5
};
```

Constant descriptions

`kGetAtomicInstNoExpandedSamples`
: Eliminate the expanded samples.

`kGetAtomicInstNoOriginalSamples`
: Eliminate the original samples.

`kGetAtomicInstNoSamples`
: Eliminate both the original and expanded samples.

`kGetAtomicInstNoKnobList`
: Eliminate the knob list.

`kGetAtomicInstNoInstrumentInfo`
: Eliminate the About box information.

`kGetAtomicInstOriginalKnobList`
: Include the original knob list.

`kGetAtomicInstAllKnobs`
: Include the current knob list.

Setting Atomic Instruments

These flags specify details of initializing a part with an atomic instrument and are passed to the `MusicSetPartAtomicInstrument` function.

```
enum {
    kSetAtomicInstKeepOriginalInstrument    = 1 << 0,
    kSetAtomicInstShareAcrossParts          = 1 << 1,
    kSetAtomicInstCallerTosses              = 1 << 2,
    kSetAtomicInstDontPreprocess            = 1 << 7
};
```

Constant descriptions

`kSetAtomicInstKeepOriginalInstrument`
Keep original sample after expansion.

`kSetAtomicInstShareAcrossParts`
Remove the instrument when the application quits.

`kSetAtomicInstCallerTosses`
The caller isn't keeping a copy of the atomic instrument for later calls to `NASetAtomicInstrument`.

`kSetAtomicInstDontPreprocess`
Don't expand the sample. You would only set this bit if you know the instrument is digitally clean or you got it from a `MusicGetPartAtomicInstrument` call.

Instrument Info Flags

Use these flags in the `MusicGetInstrumentInfo` function (page 19-118) to indicate which instruments and instrument names you are interested in.

```
enum {
    kGetInstrumentInfoNoBuiltIn         = 1 << 0,
    kGetInstrumentInfoMidiUserInst      = 1 << 1,
    kGetInstrumentInfoNoIText           = 1 << 2
};
```

Constant descriptions

`kGetInstrumentInfoNoBuiltIn`
Don't return built-in instruments.

`kGetInstrumentInfoMidiUserInst`
: Do return user instruments for a MIDI device.

`kGetInstrumentInfoNoIText`
: Don't return international text strings.

Synthesizer Connection Type Flags

These flags provide information about a MIDI device's connection and are used in the synthesizer connections structure (page 19-57).

```
enum {
    kSynthesizerConnectionMono      = 1,
    kSynthesizerConnectionMMgr      = 2,
    kSynthesizerConnectionOMS       = 4,
    kSynthesizerConnectionQT        = 8,
    kSynthesizerConnectionFMS       = 16
};
```

Constant descriptions

`kSynthesizerConnectionMono`
: If set, and the synthesizer can be both monophonic and polyphonic, the synthesizer is instructed to take up its channels sequentially from the system channel in monophonic mode.

`kSynthesizerConnectionMMgr`
: This connection is imported from the MIDI Manager.

`kSynthesizerConnectionOMS`
: This connection is imported from OMS.

`kSynthesizerConnectionQT`
: This connection is a QuickTime-only port.

`kSynthesizerConnectionFMS`
: This connection is imported from FMS.

Instrument Match Flags

These flags are returned in the `instMatch` field of the General MIDI information structure (page 19-54) to specify how QuickTime Music Architecture matched an instrument request to an instrument.

```
enum {
    kInstrumentExactMatch       = 0x00020000,
    kRecommendedSubstitute      = 0x00010000,
    kQualityField               = 0xFF000000,
    kRoland8BitQuality          = 0x05000000
};
typedef InstrumentAboutInfo *InstrumentAboutInfoPtr;
typedef InstrumentAboutInfoPtr *InstrumentAboutInfoHandle;
```

Constant descriptions

`kInstrumentExactMatch`
The instrument exactly matches the request.

`kInstrumentRecommendedSubstitute`
The instrument is the approved substitute.

`kInstrumentQualityField`
The quality of the selected instrument.

`kInstrumentRoland8BitQuality`
The quality of a built-in instrument. Built-in instrument quality is 5 on a scale of 0–255.

Note Request Constants

These flags specify what to do if the exact instrument requested is not found. They are used in the `flags` field of the note request information structure (page 19-58).

```
enum {
    kNoteRequestNoGM            = 1,
    kNoteRequestNoSynthType     = 2
};
```

Constant descriptions

`kNoteRequestNoGM` Don't use a General MIDI synthesizer.

`kNoteRequestNoSynthType`
Don't use another synthesizer of the same type but with a different name.

Pick Instrument Flags

The pick instrument flags provide information to the `NAPickInstrument` (page 19-91) and `NAPickEditInstrument` (page 19-93) functions on which instruments to present for the user to choose from.

```
enum {
    kPickDontMix                = 1,
    kPickSameSynth              = 2,
    kPickUserInsts              = 4,
    kPickEditAllowPick          = 16
};
```

Constant descriptions

`kPickDontMix` Show either all drum kits or all instruments depending on the current instrument. For example, if it's a drum kit, show only drum kits.

`kPickSameSynth` Show only instruments from the current synthesizer.

`kPickUserInsts` Show modifiable instruments in addition to ROM instruments.

`kPickEditAllowPick`
Present the instrument picker dialog. Used only with the `NAPickEditInstrument` function.

Note Allocator Type

Use this constant to specify the QuickTime note allocator component.

```
enum {
    kNoteAllocatorType          = 'nota'
    kNoteAllocatorComponentType = 'not2'
};
```

Constant description

`kNoteAllocatorType`
The QuickTime Music Architecture note allocator type.

Tune Queue Depth

This constant represents the maximum number of segments that can be queued with the `TuneQueue` function.

```
enum {
    kTuneQueueDepth     = 8
};
```

Constant description

`kTuneQueueDepth` Deepest you can queue tune segments.

Tune Player Type

Use this constant to specify the QuickTime tune player component.

```
enum {
    kTunePlayerType         = 'tune'
};
```

Constant descriptions

`kTunePlayerType` The QuickTime Music architecture tune player component type.

Tune Queue Flags

Use these flags in the `TuneQueue` function (page 19-63) to give detail about how to handle the queued tune.

```
enum {
    kTuneStartNow               = 1,
    kTuneDontClipNotes          = 2,
    kTuneExcludeEdgeNotes       = 4,
    kTuneQuickStart             = 8,
    kTuneLoopUntil              = 16,
    kTuneStartNewMaster         = 16384
};
```

Constant descriptions

`kTuneStartNow` Play even if another tune is playing.

`kTuneDontClipNotes`
Allow notes to finish their durations outside sample.

`kTuneExcludeEdgeNotes`
Don't play notes that start at end of tune.

`kTuneQuickStart` Leave all the controllers where they are and ignore start time.

`kTuneLoopUntil` Loop a queued tune if there is nothing else in the queue.

`kTuneStartNewMaster`
Start a new master reference timer.

Data Structures

This section describes the data structures provided by QuickTime Music Architecture.

Instrument Knob Record

An instrument knob record contains information about an instrument knob. It is defined by the `InstKnobRec` data type.

```
struct InstKnobRec {
    long                        number;
    long                        value;
};
typedef struct InstKnobRec InstKnobRec;
```

Field descriptions

`number` A knob ID or index. A nonzero value in the high byte indicates that it is an ID. The knob index ranges from 1 to the number of knobs; the ID is an arbitrary number.

`value` The value the knob is set to.

Instrument Knob List

An instrument knob list contains a list of sound parameters. It is defined by the `InstKnobList` data type.

```
struct InstKnobList {
    long                        knobCount;
    long                        knobFlags;
    InstKnobRec                 knob[1];
};
typedef struct InstKnobList InstKnobList;
```

Field descriptions

`knobCount`	The number of instrument knob records in the list.
`knobFlags`	Instructions on what to do if a requested knob is not in the list. See "Instrument Knob Flags" on page 30.
`InstKnobRec`	An array of instrument knob records.

Atomic Instrument Sample Description Record

A sample description record contains a description of an audio sample, including sample rate, loop points, and lowest and highest key to play on. It is defined by the `InstSampleDescRec` data type.

```
struct InstSampleDescRec {
    OSType                      dataFormat;
    short                       numChannels;
    short                       sampleSize;
    UnsignedFixed               sampleRate;
    short                       sampleDataID;
    long                        offset;
    long                        numSamples;
    long                        loopType;
    long                        loopStart;
    long                        loopEnd;
    long                        pitchNormal;
    long                        pitchLow;
    long                        pitchHigh;
};
typedef struct InstSampleDescRec InstSampleDescRec;
```

Field descriptions

`dataFormat`	The data format type. This is either `'twos'`, for signed data, or `'raw '`, for unsigned data.
`numChannels`	The number of channels of data present in the sample.
`sampleSize`	The size of the sample— 8-bit or 16-bit.
`sampleRate`	The rate at which to play the sample in unsigned fixed-point 16.16.
`sampleDataID`	The ID number of a sample data atom that contains the sample audio data.
`offset`	Set to 0.
`numSamples`	The number of data samples in the sound.
`loopType`	The type of loop. See “Loop Type Constants” on page 31.
`loopStart`	Indicates the beginning of the portion of the sample that is looped if the sound is sustained. The position is given in the number of data samples from the start of the sound.
`loopEnd`	Indicates the end of the portion of the sample that is looped if the sound is sustained. The position is given in the number of data samples from the start of the sound.
`pitchNormal`	The number of the MIDI note produced if the sample is played at the rate specified in `sampleRate`.
`pitchLow`	The lowest pitch at which to play the sample. Use for instruments, such as pianos, that have different samples to use for different pitch ranges.
`pitchHigh`	The highest pitch at which to play the sample. Use for instruments, such as pianos, that have different samples to use for different pitch ranges.

Synthesizer Description Structure

A synthesizer description structure contains information about a synthesizer. It is defined by the `SynthesizerDescription` data type.

```
struct SynthesizerDescription {
    OSType                      synthesizerType;
    Str31                       name;
    unsigned long               flags;
    unsigned long               voiceCount;
```

```
    unsigned long               partCount;
    unsigned long               instrumentCount;
    unsigned long               modifiableInstrumentCount;
    unsigned long               channelMask;
    unsigned long               drumPartCount;
    unsigned long               drumCount;
    unsigned long               modifiableDrumCount;
    unsigned long               drumChannelMask;
    unsigned long               outputCount;
    unsigned long               latency;
    unsigned long               controllers[4];
    unsigned long               gmInstruments[4];
    unsigned long               gmDrums[4];
};
typedef struct SynthesizerDescription SynthesizerDescription;
```

Field descriptions

`synthesizerType` The synthesizer type. This is the same as the music component subtype.

`name` Text name of the synthesizer type.

`flags` Various information about how the synthesizer works. See "Synthesizer Description Flags" on page 32.

`voiceCount` Maximum polyphony.

`partCount` Maximum multi-timbrality (and MIDI channels).

`instrumentCount` The number of built-in ROM instruments. This does not include General MIDI instruments.

`modifiableInstrumentCount` The number of slots available for saving user-modified instruments.

`channelMask` Which channels a MIDI device always uses for instruments. Set to `FFFF` for all channels.

`drumPartCount` The maximum multi-timbrality of drum parts. For synthesizers where drum kits are separated from instruments.

`drumCount` The number of built-in ROM drum kits. This does not include General MIDI drum kits. For synthesizers where drum kits are separated from instruments

`modifiableDrumCount`
: The number of slots available for saving user-modified drum kits. For MIDI synthesizers where drum kits are separated from instruments

`drumChannelMask`
: Which channels a MIDI device always uses for drum kits. Set to `FFFF` for all channels

`outputCount`
: The number of audio outputs. This is usually two.

`latency`
: Response time in µSec.

`controllers[4]`
: An array of 128 bits identifying the available controllers. See "Controller Numbers" on page 33. Bits are numbered from 1 to 128, starting with the most significant bit of the longword, and continuing to the least significant of the last bit.

`gmInstruments[4]`
: An array of 128 bits giving the available General MIDI instruments.

`gmDrums[4]`
: An array of 128 bits giving the available General MIDI drum kits.

Tone Description Structure

A tone description structure provides the information needed to produce a specific musical sound. The tune header has a tone description for each instrument used. Tone descriptions are also used in the tone description atoms of atomic instruments. The tone description structure is defined by the `ToneDescription` data type.

```
struct ToneDescription {
    OSType                      synthesizerType;
    Str31                       synthesizerName;
    Str31                       instrumentName;
    long                        instrumentNumber;
    long                        gmNumber;
};
typedef struct ToneDescription ToneDescription;
```

Field descriptions

`synthesizerType`
: The synthesizer type. See "Synthesizer Type Constants" (page 19-31) for two possible types. Set to zero if any type of synthesizer is acceptable.

`synthesizerName`	Name of this instantiation of the synthesizer. Set to zero if the name is unimportant.
`instrumentName`	The name of the instrument.
`instrumentNumber`	The instrument number for non–General MIDI instruments.
`gmNumber`	Best matching General MIDI number to use if the `instrumentNumber` number in is not found. If you don't provide a General MIDI number and neither the instrument name nor the instrument number is found, the tone plays nothing.

Knob Description Record

A knob description record contains sound parameter values for a single knob. It is defined by the `KnobDescription` data type.

```
struct KnobDescription {
    Str63                       name;
    long                        lowValue;
    long                        highValue;
    long                        defaultValue;
    long                        flags;
    long                        knobID;
};
typedef struct KnobDescription KnobDescription;
```

Field descriptions

`name`	The name of the knob.
`lowValue`	The lowest number you can set the knob to.
`highValue`	The highest number you can set the knob to.
`defaultValue`	A value to use for the default.
`flags`	Various information about the knob. See "Knob Flags" on page 37.
`knobID`	A knob ID or index. A nonzero value in the high byte indicates that it is an ID. The knob index ranges from 1 to the number of knobs; the ID is an arbitrary number. Use the knob ID to refer to the knob in preference over the knob index, which may change.

Instrument About Information

The instrument About information structure contains the information that appears in the instrument's About box and is returned by the `MusicGetInstrumentAboutInfo` function (page 19-118). It is defined by the `InstrumentAboutInfo` data type.

```
struct InstrumentAboutInfo {
    PicHandle                   p;
    Str255                      author;
    Str255                      copyright;
    Str255                      other;
};
typedef struct InstrumentAboutInfo InstrumentAboutInfo;
```

Field descriptions

`p`	A handle to a graphic for the About box.
`author`	The author's name.
`copyright`	The copyright information.
`other`	Any other textual information.

MIDI Packet

The MIDI packet structure describes the data passed by note allocation calls. It is defined by the `MusicMIDIPacket` data type.

```
struct MusicMIDIPacket {
    unsigned short          length;
    unsigned long           reserved;
    UInt8                   data[249];
};
typedef struct MusicMIDIPacket MusicMIDIPacket;
```

Field descriptions

`length`	The length of the data in the packet.
`reserved`	This field contains zero or one of the music packet status constants. See "Music Packet Status" on page 39.
`data[249]`	The MIDI data.

Note
This is the count of data bytes only, unlike MIDI Manager or OMS packets.

Instrument Information Record

The instrument information record provides identifiers for instruments and is part of the instrument information list. It is defined by the `InstrumentInfoRecord` data type.

```
struct InstrumentInfoRecord {
    long                        instrumentNumber;
    long                        flags;
    long                        toneNameIndex;
    long                        itxtNameAtomID;
};
typedef struct InstrumentInfoRecord InstrumentInfoRecord;
```

Field descriptions

`instrumentNumber` The instrument number. If the number is 0, the name is an instrument category.

`flags` Unused. Must be zero

`toneNameIndex` The instrument's position in the `toneNames` index stored in the instrument information list this record is a part of. The index is a one-based index.

`itxtNameAtomID` The instrument's position in the `itxtNames` index stored in the instrument information list this record is a part of.

Instrument Information List

An instrument information list contains the list of instruments available on a synthesizer. It is defined by the `InstrumentInfoList` data type.

```
struct InstrumentInfoList {
    long                        recordCount;
    Handle                      toneNames;
    QTAtomContainer             itxtNames;
    InstrumentInfoRecord        info[1];
};
```

```
typedef struct InstrumentInfoList InstrumentInfoList;
typedef InstrumentInfoList *InstrumentInfoListPtr;
typedef InstrumentInfoListPtr *InstrumentInfoListHandle;
```

Field descriptions

`recordCount`	The number of records in the list.
`toneNames`	A string list of the instrument names as specified in their tone descriptions.
`itxtNames`	A list of international text names, taken from the name atoms.
`info[1]`	An array of instrument information records.

General MIDI Instrument Information Structure

The General MIDI information structure provides information about a General MIDI instrument within an instrument component. It is defined by the `GMInstrumentInfo` data type.

```
struct GMInstrumentInfo {
    long                        cmpInstID;
    long                        gmInstNum;
    long                        instMatch;
};
typedef struct GMInstrumentInfo GMInstrumentInfo;
typedef GMInstrumentInfo *GMInstrumentInfoPtr;
typedef GMInstrumentInfoPtr *GMInstrumentInfoHandle;
```

Field descriptions

`cmpInstID`	The number of the instrument within the instrument component.
`gmInstNum`	The General MIDI, or standard, instrument number.
`instMatch`	A flag indicating how the instrument matches the requested instrument. See “Instrument Match Flags” on page 42.

Non-General MIDI Instrument Information Record

The non–General MIDI information record provides information about non-General MIDI instruments within an instrument component. It is defined by the `nonGMInstrumentInfoRecord` data type.

```
struct nonGMInstrumentInfoRecord {
    long                        cmpInstID;
    long                        flags;
    long                        toneNameIndex;
    long                        itxtNameAtomID;
};
typedef struct nonGMInstrumentInfoRecord nonGMInstrumentInfoRecord;
```

Field descriptions

`cmpInstID`	The number of the instrument within the instrument component. If the ID is 0, the name is a category name.
`flags`	Not used.
`toneNameIndex`	The instrument's position in the `toneNames` index stored in the instrument information list this record is a part of. The index is a one-based index.
`itxtNameAtomID`	The instrument's position in the `itxtNames` index stored in the instrument information list this record is a part of.

Non–General MIDI Instrument Information List

A non–General MIDI instrument information list contains the list of non–General MIDI instruments supported by an instrument component. It is defined by the `nonGMInstrumentInfo` data type.

```
struct nonGMInstrumentInfo {
    long                        recordCount;
    Handle                      toneNames;
    QTAtomContainer             itxtNames;
    nonGMInstrumentInfoRecord   instInfo[1];
};
typedef struct nonGMInstrumentInfo nonGMInstrumentInfo;
typedef nonGMInstrumentInfo *nonGMInstrumentInfoPtr;
typedef nonGMInstrumentInfoPtr *nonGMInstrumentInfoHandle;
```

Field descriptions

`recordCount`	Number of records in the list.
`toneNames`	A short string list of the instrument names as specified in their tone descriptions.
`itxtNames`	A list of international text names, taken from the name atoms.
`instInfo[1]`	An array of non–General MIDI instrument information records.

Complete Instrument Information List

The complete instrument information list contains a list of all atomic instruments supported by an instrument component. It is defined by the `InstCompInfo` data type.

```
struct InstCompInfo {
    long                        infoSize;
    long                        GMinstrumentCount;
    GMInstrumentInfoHandle      GMinstrumentInfo;
    long                        GMdrumCount;
    GMInstrumentInfoHandle      GMdrumInfo;
    long                        nonGMinstrumentCount;
    nonGMInstrumentInfoHandle   nonGMinstrumentInfo;
    long                        nonGMdrumCount;
    nonGMInstrumentInfoHandle   nonGMdrumInfo;
};
typedef struct InstCompInfo InstCompInfo;
typedef InstCompInfo *InstCompInfoPtr;
typedef InstCompInfoPtr *InstCompInfoHandle;
```

Field descriptions

`infoSize`	The size of this record in bytes.
`GMinstrumentCount`	The number of General MIDI instruments.
`GMinstrumentInfo`	A handle to a list of General MIDI instrument information records.
`GMdrumCount`	The number of General MIDI drum kits.
`GMdrumInfo`	A handle to a list of General MIDI instrument information records.

`nonGMinstrumentCount`
The number of non–General MIDI instruments.

`nonGMinstrumentInfo`
A handle to the list of non–General MIDI instruments.

`nonGMdrumCount` The number of non–General MIDI drum kits.

`nonGMdrumInfo` A handle to the list of non–General MIDI drum kits.

Synthesizer Connections for MIDI Devices

The synthesizer connection structure describes how a MIDI device is connected to the computer. It is defined by the `SynthesizerConnections` data type.

```
struct SynthesizerConnections {
    OSType                      clientID;
    OSType                      inputPortID;
    OSType                      outputPortID;
    long                        midiChannel;
    long                        flags;
    long                        unique;
    long                        reserved1;
    long                        reserved2;
};
typedef struct SynthesizerConnections SynthesizerConnections;
```

Field descriptions

`clientID` The client ID provided by the MIDI Manager or `'OMS '` for an OMS port.

`inputPortID` The ID provided by the MIDI Manager or OMS for the port used to send to the MIDI synthesizer.

`outputPortID` The ID provided by the MIDI Manager or OMS for the port that receives from a keyboard or other control device.

`midiChannel` The system MIDI channel or, for a hardware device, the slot number.

`flags` Information about the type of connection. See "Synthesizer Connection Type Flags" on page 42.

`unique` A unique ID you can use instead of an index to identify the synthesizer to the note allocator.

`reserved1` Reserved. Set to 0.

reserved2	Reserved. Set to 0.

QuickTime MIDI Port

This structure provides information about the port used by the QuickTime Music Synthesizer. The QuickTime MIDI port structure is defined by the QTMIDIPort data type.

```
struct QTMIDIPort {
    SynthesizerConnections      portConnections;
    Str63                       portName;
};
typedef struct QTMIDIPort QTMIDIPort;
```

Field descriptions

portConnections	A synthesizer connections structure (page 19-57).
portName	The name of the output port.

Note Request Information Structure

The note request information structure contains information for allocating a note channel that's additional to that included in a tone description structure. It is defined by the NoteRequestInfo data type.

```
struct NoteRequestInfo {
    UInt8                       flags;
    UInt8                       reserved;
    short                       polyphony;
    Fixed                       typicalPolyphony;
};
typedef struct NoteRequestInfo NoteRequestInfo;
```

Field descriptions

flags	Specifies what to do if the exact instrument requested in a tone description structure is not found. See "Note Request Constants" on page 43.
reserved	Reserved. Set to 0.
polyphony	Maximum number of voices.
typicalPolyphony	Hint for level mixing.

Note Request Structure

A note request structure combines a tone description structure and a note request information structure to provide all the information available for allocating a note channel. It is defined by the `NoteRequest` data type.

```
struct NoteRequest {
    NoteRequestInfo             info;
    ToneDescription             tone;
};
typedef struct NoteRequest NoteRequest;
```

Field descriptions

`info`	A note request information structure (page 19-58).
`tone`	A tone description structure (page 19-50).

Tune Status

The tune status structure provides information on the currently playing tune.

```
struct TuneStatus {
    unsigned long           *tune;
    unsigned long           *tunePtr;
    TimeValue               time;
    short                   queueCount;
    short                   queueSpots;
    TimeValue               queueTime;
    long                    reserved[3];
};
typedef struct TuneStatus TuneStatus;
```

Field descriptions

`tune`	The currently playing tune.
`tunePtr`	Current position within the playing tune.
`time`	Current tune time.
`queueCount`	Number of tunes queued up.
`queueSpots`	Number of tunes that can be added to the queue.
`queueTime`	Total amount of playing time represented by tunes in the queue. This value can be very inaccurate.

`reserved[3]` Reserved. Set to 0.

Functions

The functions provided by the note allocator component, the tune player component, music components, and instrument components are described in the following sections.

Tune Player Functions

This section describes the functions the tune player provides for setting, queueing, and manipulating music sequences. It also describes tune player utility functions.

TuneSetHeader

The `TuneSetHeader` function prepares the tune player to accept subsequent music event sequences by defining one or more parts to be used by sequence Note events.

```
pascal ComponentResult TuneSetHeader(
                    TunePlayer tp,
                    unsigned long *header);
```

`tp` You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`header` A pointer to a list of instruments that will be used in subsequent calls to the `TuneQueue` function. The list can include note request General events with subtypes of `kGeneralEventNoteRequest`, `kGeneralEventPartKey`, `kGeneralEventAtomicInstrument`, `kGeneralEventMIDIChannel`, and `kGeneralEventUsedNotes`. It can also include atomic instruments. The list is terminated by a Marker event of subtype End.

function result A result code.

DISCUSSION

The `TuneSetHeader` function is the first QuickTime Music Architecture call to play a music sequence. The `header` parameter points to one or more initialized General events and atomic instruments. The event list pointed to by the `header` parameter must conclude with a Marker event of subtype End.

Only one call to `TuneSetHeader` is required. Each `TuneSetHeader` call resets the tune player.

SEE ALSO

The `TuneSetHeaderWithSize` function (page 19-61) and the `TuneSetNoteChannels` function (page 19-62).

TuneSetHeaderWithSize

The `TuneSetHeaderWithSize` function is like the `TuneSetHeader` function in that it prepares the tune player to accept subsequent music event sequences by defining one or more parts to be used by sequence Note events. But unlike the `TuneSetHeader` function, `TuneSetHeaderWithSize` allows you to specify the header length in bytes. This prevents the call from parsing off the end if the music event sequence is missing an end marker.

```
extern pascal ComponentResult TuneSetHeaderWithSize(
                    TunePlayer tp,
                    unsigned long *header,
                    unsigned long size)
```

`tp` You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`header` A pointer to a list of instruments that will be used in subsequent calls to the `TuneQueue` function. The list can include General events with subtypes of `kGeneralEventNoteRequest`, `kGeneralEventPartKey`, `kGeneralEventAtomicInstrument`,

kGeneralEventMIDIChannel, and kGeneralEventUsedNotes. It can also include atomic instruments. The list is terminated by a Marker event of subtype End.

size — The size of the header in bytes.

function result — A result code.

SEE ALSO

The TuneSetHeader function (page 19-60) and the TuneSetNoteChannels function (page 19-62).

TuneSetNoteChannels

The TuneSetNoteChannels function prepares the tune player to accept music event sequences by allocating specified note channels for them. It is an alternative to the TuneSetHeader function.

```
extern pascal ComponentResult TuneSetNoteChannels(
                    TunePlayer tp,
                    unsigned long count,
                    NoteChannel* noteChannelList,
                    TunePlayCallBackUPP playCallBackProc,
                    long refCon)
```

tp — You obtain the tune player identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

count — The number of note channels to be set.

noteChannelList — A pointer to a list of note channel identifiers. You obtain the note channel identifiers from the NANewNoteChannel and the NANewNoteChannelFromAtomicInstrument functions.

playCallBackProc — A function that is called back for each event that is played.

refCon A reference constant passed to the callback function.

function result A result code.

SEE ALSO

The TuneSetHeader function (page 19-60) and the TuneSetHeaderWithSize function (page 19-61).

TuneQueue

The TuneQueue function places a sequence of music events into a queue to be played.

```
pascal ComponentResult TuneQueue(
                    TunePlayer tp,
                    unsigned long *tune,
                    Fixed tuneRate,
                    unsigned long tuneStartPosition,
                    unsigned long tuneStopPosition,
                    unsigned long queueFlags,
                    TuneCallBackUPP callBackProc,
                    long refCon)
```

tp You obtain the tune player identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

*tune Pointer to an array of events, terminated by a Marker event of subtype End.

tuneRate Fixed-point speed at which to play the sequence. "Normal" speed is 0x00010000.

tuneStartPosition
Sequence starting time.

tuneStopPosition
Sequence ending time.

queueFlags	Flags with details about how to play the queued tunes. For valid values see "Tune Queue Flags" (page 19-45).
callBackProc	Points to your callback function. Your callback function must have the following form: `pascal void MyCallBackProc (QTCallBack cb, long refcon);`
refcon	Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.
function result	A result code. In addition to QuickTime Music Architecture result codes, this function may return TimeBase result codes.

DISCUSSION

The tuneStartPosition and tuneStopPosition specify, in time units numbered from zero for the beginning of the sequence, which part of the queued sequence to play. To play all of it, pass 0 and 0xFFFFFFFF respectively.

If there is a sequence currently playing, the newly queued sequence will begin as soon as the active sequence ends unless the queueFlags parameter is kTuneStartNow, in which case the currently playing sequence will be immediately terminated and the new one started.

TuneStop

The TuneStop function stops a currently playing sequence.

```
pascal ComponentResult TuneStop(
                    TunePlayer tp,
                    long stopFlags);
```

tp	You obtain the tune player identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
stopFlags	Must be zero.

function result A result code.

TuneGetVolume

The `TuneGetVolume` function returns the volume associated with the entire sequence.

```
pascal ComponentResult TuneGetVolume(
                    TunePlayer tp);
```

`tp` You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

function result The volume as a value from 0.0 to 1.0 or a negative result code.

TuneSetVolume

The `TuneSetVolume` function sets the volume for the entire sequence.

```
pascal ComponentResult TuneSetVolume(
                    TunePlayer tp,
                    Fixed volume);
```

`tp` You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`volume` The volume to use for the sequence. The value is a fixed 16.16 number.

function result A result code.

DISCUSSION

The `TuneSetVolume` function sets the volume level of the active sequence to the value of the `volume` parameter ranging from 0.0 to 1.0.

Note
Individual instruments within the sequence can maintain independent volume levels. ◆

TuneSetSoundLocalization

The `TuneSetSoundLocalization` function passes sound localization data to a tune player.

```
extern pascal ComponentResult TuneSetSoundLocalization(
                    TunePlayer tp,
                    Handle data)
```

`tp`	You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`data`	The sound localization data to be passed.
function result	A result code.

TuneGetTimeBase

The `TuneGetTimeBase` function returns the time base of the tune player.

```
pascal ComponentResult TuneGetTimeBase(
                    TunePlayer tp,
                    TimeBase *tb);
```

`tp`	You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`*tb`	An initialized `TimeBase` object.
function result	A result code.

DISCUSSION

The `TuneGetTimeBase` function returns, in the `TimeBase` parameter, the time base used to control the sequence timing. The sequence may be controlled in several ways through its time base. The rate of playback may be changed, or the `TimeBase` object may be slaved to a clock or time base different than real time.

TuneGetTimeScale

The `TuneGetTimeScale` function returns the current time scale, in units-per-second, for the specified tune player instance.

```
pascal ComponentResult TuneGetTimeScale(
                    TunePlayer tp,
                    TimeScale *scale);
```

`tp`	You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`*scale`	An initialized `TimeScale` object.
function result	A result code.

TuneSetTimeScale

The `TuneSetTimeScale` function sets the time scale used by the specified tune player instance.

```
pascal ComponentResult TuneSetTimeScale(
                    TunePlayer tp,
                    TimeScale scale);
```

`tp` You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`scale` The time scale value to be used, in units-per-second.

function result A result code.

DISCUSSION

The `TuneSetTimeScale` function sets the time scale data used by the tune player's sequence data when interpreting time based events.

TuneInstant

You can use the `TuneInstant` function to play the particular sequence events active at a specified position.

```
pascal ComponentResult TuneInstant(
                    TunePlayer tp,
                    unsigned long *tune,
                    long tunePosition)
```

`tp` You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`*tune` Pointer to tune sequence data.

`tunePosition` Position within tune sequence data in time units.

function result A result code.

DISCUSSION

The `TuneInstant` function plays the notes that are "on" at the point specified by the `tunePosition` parameter. The notes are started and then left playing on return. The notes may be silenced by calling the `TuneStop` function. This call is useful for enabling user "scrubbing" on a sequence.

TunePreroll

The `TunePreroll` function prepares for playing tune player sequence data by attempting to reserve note channels for each part in the sequence.

```
pascal ComponentResult TunePreroll (TunePlayer tp);
```

`tp` You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

function result A result code.

TuneUnroll

The `TuneUnroll` function releases any note channel resources that may have been locked down by previous calls to `TunePreroll` for this tune player.

```
pascal ComponentResult TuneUnroll (TunePlayer tp);
```

`tp` You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

function result A result code.

TuneGetIndexedNoteChannel

You can use the `TuneGetIndexedNoteChannel` function to determine how many parts the tune is playing and which instrument is assigned to those parts.

```
pascal ComponentResult TuneGetIndexedNoteChannel(
                    TunePlayer tp,
                    short i,
                    NoteChannel *nc);
```

`tp`	You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`i`	Note channel index or 0 to get the number of parts.
`*nc`	Allocated initialized note channel.
function result	A positive value is the number of note channels used by the tune player; a negative value is a result code.

DISCUSSION

The tune player allocates note channels that best satisfy the requested instrument in the tune header. The application may use this call to determine which instrument was actually used for each note channel. The `TuneGetIndexedNoteChannel` function takes a tune player in the `tp` parameter and returns the number of parts (1...n) allocated to the tune player. You can then pass the function a part index and it returns, in the `nc` parameter, the note channel allocated for that part.

TuneGetStatus

The `TuneGetStatus` function returns an initialized structure describing the state of the tune player instance.

```
pascal ComponentResult TuneGetStatus(
                    TunePlayer tp,
                    TuneStatus *status);
```

tp	You obtain the tune player identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
*status	A pointer to an initialized tune status structure (page 19-59).
function result	A result code.

TuneSetPartTranspose

The TuneSetPartTranspose function modifies the pitch and volume of every note of a tune.

```
extern pascal ComponentResult TuneSetPartTranspose(
                    TunePlayer tp,
                    unsigned long part,
                    long transpose,
                    long velocityShift)
```

tp	You obtain the tune player identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
part	The part for which you want to change pitch and volume.
transpose	A value by which to modify the pitch of the note. The value is a small integer for semitones or an 8.8 fixed-point number for microtones.
velocityShift	A value to add to the velocity parameter passed to the NAPlayNote function.
function result	A result code.

TuneGetNoteAllocator

The `TuneGetNoteAllocator` function returns the instance of the note allocator that the tune player is using.

```
extern pascal NoteAllocator TuneGetNoteAllocator (TunePlayer tp)
```

`tp` You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

function result A note allocator or a result code.

TuneSetSofter

The `TuneSetSofter` function adjusts the volume a tune is played at to the softer volume produced by QuickTime 2.1. Files imported with QuickTime 2.1 automatically played softer. Files imported with QuickTime 2.5 play at the new, louder volume.

```
extern pascal ComponentResult TuneSetSofter(
                    TunePlayer tp,
                    long softer)
```

`tp` You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`softer` A value of 1 means play at the QuickTime 2.1 volume; a value of 0 means don't make the volume softer.

function result A result code.

TuneSetBalance

Use the `TuneSetBalance` function to modify the pan controller setting for a tune player.

```
extern pascal ComponentResult TuneSetBalance(
                    TunePlayer tp,
                    long balance)
```

`tp` You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`balance` Modifies the pan controller setting. Valid values are between –128 to 128 for left to right balance.

function result A result code.

TuneTask

Call the `TuneTask` function periodically to allow a tune player to perform tasks it must perform at foreground task time.

```
extern pascal ComponentResult TuneTask (TunePlayer tp)
```

`tp` You obtain the tune player identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

function result A result code.

DISCUSSION

Certain operations can be performed only at foreground application task time. Specifically, the QuickTime Music Synthesizer cannot load instruments from disk at interrupt time. As a result, embedded program changes are not performed until `TuneTask` is called.

Note Allocator Functions: Note Channel Allocation and Use

The functions described in this section create, manipulate, and get information about note channels.

NANewNoteChannel

The `NANewNoteChannel` function requests a new note channel with the qualities described in the `noteRequest` structure.

```
pascal ComponentResult NANewNoteChannel(
                    NoteAllocator ci,
                    NoteRequest *noteRequest,
                    NoteChannel *outChannel);
```

`ci`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`noteRequest`	A pointer to a note request structure.
`outChannel`	On exit, a pointer to an identifier for a new note channel or NIL if the function fails to create a note channel.
function result	A result code.

DISCUSSION

The caller may request an instrument that is not currently allocated to a part. In that case, the `NANewNoteChannel` function may return a value in `outChannel`, even though the request cannot initially be satisfied. The note channel may become valid at a later time, as other note channels are released or other music components are registered.

If an error occurs the note `noteChannel` will be initialized to NIL.

NANewNoteChannelFromAtomicInstrument

You can use the `NANewNoteChannelFromAtomicInstrument` function to request a new note channel for an atomic instrument.

```
extern pascal ComponentResult NANewNoteChannelFromAtomicInstrument(
                    NoteAllocator ci,
                    AtomicInstrumentPtr instrument,
                    long flags,
                    NoteChannel *outChannel)
```

`ci`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`instrument`	A pointer to the atomic instrument. This may be a dereferenced locked QT atom container.
`flags`	These flags specify details of initializing a part with an atomic instrument. See "Setting Atomic Instruments" on page 41.
`outChannel`	On exit, a pointer to an identifier for a new note channel or NIL if the function fails to create a note channel.
function result	A result code.

DISCUSSION

The `NANewNoteChannelFromAtomicInstrument` function takes a note allocator identifier in the `ci` parameter and a pointer to the atomic instrument you are requesting a new channel for in the `instrument` parameter. Among other things, you can specify how to handle the expanded sample with the `flags` parameter.

The function returns the note channel allocated for the instrument in the `outChannel` parameter or NIL if an error occurs.

NADisposeNoteChannel

The `NADisposeNoteChannel` function deletes the specified note channel.

```
pascal ComponentResult NADisposeNoteChannel(
                    NoteAllocator ci,
                    NoteChannel noteChannel);
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`noteChannel` Note channel to be disposed. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

function result A result code.

NAGetNoteChannelInfo

The `NAGetNoteChannelInfo` function returns the index of the music component for the allocated channel and its part number on that music component.

```
pascal ComponentResult NAGetNoteChannelInfo(
                    NoteAllocator ci,
                    NoteChannel noteChannel,
                    long *index,
                    long *part)
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`noteChannel` Note channel to get information about. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

`*index` Music component index.

`*part`	Music component part pointer.
function result	A result code.

DISCUSSION

The `NAGetNoteChannelInfo` function allows direct access to the music component allocated to the note channel by the note allocator. The index returned will be invalid if music components are subsequently registered or unregistered.

NAGetIndNoteChannel

The `NAGetIndNoteChannel` function returns the number of note channels handled by the specified note allocator instance. It can also return a requested note channel.

```
extern pascal ComponentResult NAGetIndNoteChannel(
                    NoteAllocator ci,
                    long index,
                    NoteChannel *nc,
                    long *seed)
```

`ci`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`index`	The index of the note channel to get or 0 to get the total number of note channels handled by the note allocator.
`nc`	The note channel requested.
`seed`	A number that changes on successive calls if anything significant changes about a note channel—for example, if the note channel has been reallocated or released.
function result	Positive results are the index count; negative results are error codes.

DISCUSSION

To get a count of the note channels pass the `NAGetIndNoteChannel` function 0 in the `index` parameter. To get a specific note channel, pass the index value returned by a previous call to `NAGetIndNoteChannel`.

NAUseDefaultMIDIInput

The `NAUseDefaultMIDIInput` function defines an entry point to service external MIDI device events.

```
pascal ComponentResult NAUseDefaultMIDIInput (
                    NoteAllocator ci,
                    MusicMIDIReadHookUPP readHook,
                    long refCon,
                    unsigned long flags)
```

`ci`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`readHook`	Process pointer for MIDI service.
`refcon`	Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.
`flags`	Must contain zero.
function result	A result code.

DISCUSSION

The `NAUseDefaultMIDIInput` function specifies an application's procedure to service external MIDI events. The specified application's procedure call, defined by `readHook`, will be called when the external default MIDI device has incoming MIDI data for the application.

NALoseDefaultMIDIInput

The `NALoseDefaultMIDIInput` function removes the external default MIDI service procedure call, if previously defined by `NAUseDefaultMIDIInput`.

```
pascal ComponentResult NALoseDefaultMIDIInput (NoteAllocator ci);
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

function result A result code or `-1` if a default MIDI device was not in use.

NAPrerollNoteChannel

The `NAPrerollNoteChannel` function attempts to reallocate the note channel, if it was invalid previously.

```
pascal ComponentResult NAPrerollNoteChannel(
                    NoteAllocator ci,
                    NoteChannel noteChannel);
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`noteChannel` Note channel to be re-allocated. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

function result A result code.

DISCUSSION

The `NAPrerollNoteChannel` function attempts to reallocate the note channel, if it was invalid previously. It could have been invalid if there were no available voices on any registered music components when the note channel was created.

NAUnrollNoteChannel

The `NAUnrollNoteChannel` function marks a note channel as available to be stolen.

```
pascal ComponentResult NAUnrollNoteChannel(
                    NoteAllocator ci,
                    NoteChannel noteChannel);
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`noteChannel` Note channel to be unrolled. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

function result A result code.

NAResetNoteChannel

The `NAResetNoteChannel` function turns off all currently "on" notes on the note channel, and resets all controllers to their default values.

```
pascal ComponentResult NAResetNoteChannel(
                    NoteAllocator ci,
                    NoteChannel noteChannel);
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`noteChannel` The note channel to reset. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

function result A result code.

DISCUSSION

The `NAResetNoteChannel` function resets the specified note channel by turning "off" any note currently playing. All controllers are reset to their default state. The effects of the `NAResetNoteChannel` call are propagated down to the allocated part within the appropriate music component.

NASetNoteChannelVolume

The `NASetNoteChannelVolume` function sets the volume on the specified note channel.

```
pascal ComponentResult NASetNoteChannelVolume(
                    NoteAllocator ci,
                    NoteChannel noteChannel,
                    Fixed volume);
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`noteChannel` The note channel to reset. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

`volume` The volume to set the channel to. The value is a fixed 16.16 number.

DISCUSSION

The `NASetNoteChannelVolume` function sets the volume for the note channel, which is different than a controller 7 (volume controller) setting.

Both volume settings allow fractional values of 0.0 to 1.0. Each value will modify the other. For example, a controller value of 0.5 and a `NASetNoteChannelVolume` value of 0.5 result in a 0.25 volume level.

NASetNoteChannelBalance

The `NASetNoteChannelBalance` function modifies the pan controller setting for a note channel.

```
extern pascal ComponentResult NASetNoteChannelBalance(
                    NoteAllocator ci,
                    NoteChannel noteChannel,
                    long balance)
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`noteChannel` The note channel to be balanced. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

`balance` Specifies how to modify the pan controller setting. Valid values are between –128 to 128 for left to right balance.

function result A result code.

NASetNoteChannelSoundLocalization

The `NASetNoteChannelSoundLocalization` function passes sound localization data to a note channel.

```
extern pascal ComponentResult NASetNoteChannelSoundLocalization(
                    NoteAllocator ci,
                    NoteChannel noteChannel,
                    Handle data)
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

noteChannel	The note channel to pass the data to. You obtain the note channel identifier from the NANewNoteChannel or the NANewNoteChannelFromAtomicInstrument function.
data	Sound localization data.
function result	A result code.

NAPlayNote

The NAPlayNote function plays a note with a specified pitch and velocity on the specified note channel.

```
pascal ComponentResult NAPlayNote(
                    NoteAllocator ci,
                    NoteChannel noteChannel,
                    long pitch,
                    long velocity);
```

ci	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
noteChannel	The note channel to play the note. You obtain the note channel identifier from the NANewNoteChannel or the NANewNoteChannelFromAtomicInstrument function.
pitch	The pitch at which to play the note. You can specify values as integer pitch values (0–127 where 60 is middle C) or fractional pitch values (256 (0x1.00) through 32767 (0x7F.FF)).
velocity	The velocity with which the key is struck. A value of 0 is silence; a value of 127 is maximum force.
function result	A result code.

DISCUSSION

The NAPlayNote function plays a specific note. If the pitch is a number from 0 to 127, then it is the MIDI pitch, where 60 is middle C. If the pitch is a positive

number above 65535, then the value is a fixed-point pitch value. Thus, microtonal values may be specified. The range 256 (0x01.00) through 32767 (0x7F.FF), and all negative values, are not defined, and should not be used.

The velocity refers to how hard the key was struck (if performed on a keyboard-instrument). Typically, this translates directly to volume, but on many synthesizers this also subtly alters the timbre of the tone.

NASetController

The NASetController function changes the controller setting on a note channel to a specified value.

```
pascal ComponentResult NASetController
                    (NoteAllocator ci,
                    NoteChannel noteChannel,
                    long controllerNumber,
                    long controllerValue);
```

ci — You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

noteChannel — Note channel on which to change controller. You obtain the note channel identifier from the NANewNoteChannel or the NANewNoteChannelFromAtomicInstrument function.

controllerNumber — The controller to set. For valid values, see "Controller Numbers" (page 19-33).

controllerValue — Value for controller setting, typically 0 (0x00.00) to 32767 (0x7F.FF)

function result — A result code.

NAGetKnob

Use the `NAGetKnob` function to get the value of a knob for a given note channel.

```
extern pascal ComponentResult NAGetKnob(
                    NoteAllocator ci,
                    NoteChannel noteChannel,
                    long knobNumber,
                    long *knobValue)
```

`ci`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`noteChannel`	The note channel whose knob value you want to get. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.
`knobNumber`	The index or ID of the knob whose value you want to get.
`knobValue`	On exit, the value of the knob.
function result	A result code.

DISCUSSION

The `NAGetKnob` function takes a note allocator component identifier in the `ci` parameter, a note channel identifier in the `noteChannel` parameter, and the knob index or ID in the `knobNumber` parameter. It returns, in the `knobValue` parameter, a pointer to the current value of the knob.

NASetKnob

The NASetKnob function sets a note channel knob to a particular value.

```
pascal ComponentResult NASetKnob(
                    NoteAllocator ci,
                    NoteChannel noteChannel,
                    long knobNumber,
                    long knobValue)
```

ci	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
noteChannel	Note channel on which to set the knob value. You obtain the note channel identifier from the NANewNoteChannel or the NANewNoteChannelFromAtomicInstrument function.
knobNumber	Index or ID of the knob to be set.
knobValue	Value to set knob to.
function result	A result code.

DISCUSSION

The NASetKnob function takes a note allocator component identifier in the ci parameter, a note channel identifier in the noteChannel parameter, the knob ID or index in the knobNumber parameter, and a knob value in the knobValue parameter. It sets the specified knob to the given value.

NAFindNoteChannelTone

The `NAFindNoteChannelTone` function locates the instrument that best fits a requested tone description for a specific channel.

```
pascal ComponentResult NAFindNoteChannelTone(
                    NoteAllocator ci,
                    NoteChannel noteChannel,
                    ToneDescription *td,
                    long *instrumentNumber);
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`noteChannel` The note channel for which you want an instrument. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

`*td` Description for instrument fit.

`*instrumentNumber`
On exit, the number of the instrument that best fits the tone description.

function result A result code.

NASetInstrumentNumber

The `NASetInstrumentNumber` function initializes a synthesizer part with the specified instrument.

```
pascal ComponentResult NASetInstrumentNumber(
                    NoteAllocator ci,
                    NoteChannel noteChannel,
                    short instrumentNumber);
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`noteChannel` Note channel to initialize with the instrument. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

`instrumentNumber`
Number of the instrument to initialize the part with. This number is unique to each synthesizer. General MIDI synthesizers all share the range 1–128 and 16365 to `kLastDrumKit`.

function result A result code.

NASetInstrumentNumberInterruptSafe

You can use the `NASetInstrumentNumberInterruptSafe` function to initialize a synthesizer part with the specified instrument during interrupt time.

```
extern pascal ComponentResult NASetInstrumentNumberInterruptSafe(
                    NoteAllocator ci,
                    NoteChannel noteChannel,
                    long instrumentNumber);
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`noteChannel` Note channel to initialize with the instrument. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

`instrumentNumber`
Number of the instrument to initialize the part with.

function result A result code.

DISCUSSION

If the instrument is not already loaded when you call the `NASetInstrumentNumberInterruptSafe` function, you have to wait for the next call to the `NATask` function for the instrument to become available.

NASetAtomicInstrument

The `NASetAtomicInstrument` function initializes a synthesizer part with an atomic instrument.

```
extern pascal ComponentResult NASetAtomicInstrument(
                    NoteAllocator ci,
                    NoteChannel noteChannel,
                    AtomicInstrumentPtr instrument,
                    long flags)
```

`ci`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`noteChannel`	The note channel to apply the atomic instrument to. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.
`instrument`	A pointer to the atomic instrument. This can be a locked, dereferenced atomic instrument.
`flags`	Details about how to initialize the part. For a description of the flags, see "Setting Atomic Instruments" (page 19-41).
function result	A result code.

NASendMIDI

Use the NASendMIDI function to send a MIDI music packet to a synthesizer that contains a specific note channel.

```
extern pascal ComponentResult NASendMIDI(
                    NoteAllocator ci,
                    NoteChannel noteChannel,
                    MusicMIDIPacket *mp)
```

ci	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
noteChannel	The function sends the packet to the synthesizer that contains this note channel. You obtain the note channel identifier from the NANewNoteChannel or the NANewNoteChannelFromAtomicInstrument function.
mp	The music packet to be sent.
function result	A result code.

DISCUSSION

The NASendMIDI function sends the MIDI music packet pointed to by the mp parameter to the synthesizer that contains the note channel identified by the noteChannel parameter. The ci parameter specifies the note allocator instance to use.

NAGetNoteRequest

The NAGetNoteRequest function gets the note request passed to a note channel.

```
extern pascal ComponentResult NAGetNoteRequest(
                    NoteAllocator ci,
                    NoteChannel noteChannel,
                    NoteRequest *nrOut)
```

ci	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
noteChannel	The note channel whose note request you want to get. You obtain the note channel identifier from the NANewNoteChannel or the NANewNoteChannelFromAtomicInstrument function.
nrOut	On exit, a note request structure (page 19-59).
function result	A result code.

DISCUSSION

The NAGetNoteRequest function takes a note allocator instance in the ci parameter and a note channel identifier in the noteChannel parameter. It returns, in the *nrOut parameter, the note request that was used to allocate the specified note channel.

Note Allocator Functions: Miscellaneous Interface Tools

The functions in this section provide a user interface for instrument selection and presenting copyright information.

NAPickInstrument

The NAPickInstrument function presents a user interface for picking an instrument.

```
pascal ComponentResult NAPickInstrument(
                    NoteAllocator ci,
                    ModalFilterUPP filterProc,
                    StringPtr prompt,
                    ToneDescription *sd,
                    unsigned long flags,
                    long refCon,
                    long reserved1,
                    long reserved2)
```

`ci`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`filterProc`	Standard modal filter UPP*.
`prompt`	Dialog box prompt "New Instrument".
`*sd`	On entry, the tone description of the instrument that appears in the picker dialog. On exit, a tone description of the instrument the user selected.
`flags`	Determines whether to display the picker dialog and what instruments appear for selection. See "Pick Instrument Flags" (page 19-44).
`refcon`	Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.
`reserved1`	Must contain zero.
`reserved2`	Must contain zero.
function result	A result code or `-1` if there is a problem opening the dialog box.

DISCUSSION

The `flags` values limit which instruments appear within the dialog box. If the `kPickDontMix` flag is set, the dialog does not display a mix of synthesizer part types. For example, if the current instrument is a drum, only available drums appear in the dialog. The `kPickSameSynth` flag allows selections only within the current synthesizer. The `kPickUserInsts` flag allows user modifiable instruments to appear.

SEE ALSO

`NAPickEditInstrument` function

NAPickEditInstrument

The `NAPickEditInstrument` function presents a user interface for changing the instrument in a live note channel or modifying an atomic instrument.

```
extern pascal ComponentResult NAPickEditInstrument(
                    NoteAllocator ci,
                    ModalFilterUPP filterProc,
                    StringPtr prompt,
                    long refCon,
                    NoteChannel nc,
                    AtomicInstrument ai,
                    long flags)
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`filterProc` Standard modal filter UPP*.

`prompt` Dialog box prompt "New Instrument".

`refCon` Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.

`nc` The live note channel that appears in the dialog. If you specify a note channel, set the `ai` parameter to 0. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

`ai` The atomic instrument that appears in the dialog. If you specify an atomic instrument, set the `nc` parameter to 0. You obtain the atomic instrument from the `InstrumentGetInst` function.

`flags` Flags limiting the instruments presented. See "Pick Instrument Flags" (page 19-44)

function result A result code or `-1` if there is a problem opening the dialog.

DISCUSSION

The `flags` values limit which instruments appear within the dialog box. If the `kPickDontMix` flag is set, the dialog does not display a mix of synthesizer part types. For example, if the current instrument is a drum, only available drums appear in the dialog. The `kPickSameSynth` flag allows selections only within the current synthesizer. The `kPickUserInsts` flag allows user modifiable instruments to appear. If the `kPickEditAllowPick` flag is not set, no dialog appears.

SEE ALSO

`NAPickInstrument` function

NAStuffToneDescription

The `NAStuffToneDescription` function initializes a tone description structure with the details of a General MIDI note channel.

```
pascal ComponentResult NAStuffToneDescription(
                    NoteAllocator ci,
                    long gmNumber,
                    ToneDescription *td)
```

`ci`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`gmNumber`	A General MIDI instrument number.
`*td`	On exit, an initialized tone description. The instrument name field will be filled in with the string name for the instrument.
function result	A result code.

NAPickArrangement

The `NAPickArrangement` function displays a dialog to allow instrument selection.

```
pascal ComponentResult NAPickArrangement(
                    NoteAllocator ci,
                    ModalFilterUPP filterProc,
                    StringPtr prompt,
                    long zero1,
                    long zero2,
                    Track t,
                    StringPtr songName)
```

`ci`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`filterProc`	Standard modal filter upp*.
`prompt`	Dialog box prompt.
`zero1`	Must be 0.
`zero2`	Must be 0.
`t`	Arrangement movie track number.
`songName`	Name of song to display in dialog.
function result	A result code or `-1` if there is a problem opening the dialog.

NACopyrightDialog

The `NACopyrightDialog` function displays a copyright dialog with information specific to a music device.

```
pascal ComponentResult NACopyrightDialog(
                    NoteAllocator ci,
                    PicHandle p,
                    StringPtr author,
                    StringPtr copyright,
```

```
                    StringPtr other,
                    StringPtr title,
                    ModalFilterUPP filterProc,
                    long refCon)
```

`ci`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`p`	Picture image resource handle for dialog.
`author`	Author information.
`copyright`	Copyright information.
`other`	Any additional information.
`title`	Title information.
`filterProc`	Standard modal filter UPP*.
`refcon`	Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.
function result	A result code or –1 if there is a problem opening the dialog.

Note Allocator Functions: System Configuration and Utility

Use the functions in this section to create and maintain a database of music components, to save configuration information in the QuickTime Preferences file, to establish connections to external MIDI devices, and to allow the note allocator to perform necessary tasks at task foreground time.

NARegisterMusicDevice

The `NARegisterMusicDevice` function registers a music component with the note allocator.

```
pascal ComponentResult NARegisterMusicDevice(
                    NoteAllocator ci,
                    OSType synthType,
                    Str31 name,
                    SynthesizerConnections *connections);
```

`ci`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`synthType`	Subtype of the music component.
`name`	The synthesizer name.
`*connections`	A synthesizer connections for MIDI devices structure (page 19-57).
function result	A result code.

DISCUSSION

The value of the `synthType` parameter is the music component's subtype. The `name` parameter provides a means of distinguishing multiple instances of the same type of device and is a string that can be displayed to the user. If no value is passed in the `name` parameter, the name defaults to the name of the music component type. The name appears in the instrument picker dialog.

The `connections` parameter specifies the hardware connections to the device.

RESULT CODES

`SynthesizerErr`	If too many synthesizers registered.
`midiManagerAbsentErr`	If MIDI not available.

NAUnregisterMusicDevice

The `NAUnregisterMusicDevice` function removes a previously registered music component from the note allocator.

```
pascal ComponentResult NAUnregisterMusicDevice(
                    NoteAllocator ci,
                    long index;
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`index` Synthesizer to unregister. The value is 1 through the registered music component count returned by the `NAGetRegisteredMusicDevice` function (page 19-98).

function result A result code. In addition to quickTime Music Architecture result codes, this function may return a result code from the `CloseComponent` function.

NAGetRegisteredMusicDevice

The `NAGetRegisteredMusicDevice` function returns specifics about music components registered to the specified note allocator instance.

```
pascal ComponentResult NAGetRegisteredMusicDevice(
                    NoteAllocator ci,
                    long index,
                    OSType *synthType,
```

```
                    Str31 name,
                    SynthesizerConnections *connections,
                    MusicComponent *mc);
```

`ci`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`index`	The index of the music component to get information about or 0 to get the total number of music components registered with the note allocator.
`*synthType`	Synthesizer type.
`name`	Synthesizer name as a text string.
`*connections`	A synthesizer connections for MIDI devices structure (page 19-57).
`*mc`	Music component instance identifier.
function result	Positive values are the number of music components registered with the note allocator; negative values are result codes.

DISCUSSION

To get a count of the registered music components pass the `NAGetRegisteredMusicDevice` function 0 in the `index` parameter. The return value is the count of components. To get information about one of the music components registered with the note allocator, pass the music component index in the `index` parameter. The index value can be 1 through the number of registered components returned by a previous call to `NAGetRegisteredMusicDevice`.

If you request information about a specific registered music component, the `NAGetRegisteredMusicDevice` function returns the type of synthesizer the component supports in the `synthType` parameter, the name of the synthesizer in the `name` parameter, and the music component identifier in the `mc` parameter. For MIDI devices, it returns a pointer to a MIDI devices structure with information about the synthesizer connections.

NAGetDefaultMIDIInput

The `NASGetDefaultMIDIInput` function is used to obtain external MIDI connection information.

```
pascal ComponentResult NAGetDefaultMIDIInput(
                    NoteAllocator ci,
                    SynthesizerConnections *sc);
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`*sc` On exit, a synthesizer connections for MIDI devices structure (page 19-57).

DISCUSSION

The `NASGetDefaultMIDIInput` function returns an initialized `SynthesizerConnections` structure containing information about the external MIDI device attached to the system that has been selected as the default MIDI input device. The external MIDI device provides note input directly to the note allocator.

function result A result code.

NASetDefaultMIDIInput

The `NASetDefaultMIDIInput` function initializes an external MIDI device used to receive external note input.

```
pascal ComponentResult NASetDefaultMIDIInput(
                    NoteAllocator ci,
                    SynthesizerConnections *sc);
```

`ci`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`*sc`	A synthesizer connections for MIDI devices structure (page 19-57).

DISCUSSION

The `SynthesizerConnections` structure fields `clientID`, `inputPortID`, and `outputPortID` are MIDI Manager identifiers. The `MIDIChannel` field is the MIDI system channel value.

function result A result code.

NAGetMIDIPorts

The `NAGetMIDIPorts` function gets the MIDI input and output ports available to a note allocator.

```
extern pascal ComponentResult NAGetMIDIPorts(
                    NoteAllocator ci,
                    Handle *inputPorts,
                    Handle *outputPorts)
```

`ci`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`inputPorts`	On exit, a handle giving the number of input ports (the first two bytes) followed by a list of QuickTime MIDI port structures (page 19-58).
`outputPorts`	On exit, a handle giving the number of output ports (the first two bytes) followed by a list of QuickTime MIDI port structures (page 19-58).

function result A result code.

NASaveMusicConfiguration

The `NASaveMusicConfiguration` saves the current list of registered devices to a file.

```
pascal ComponentResult NASaveMusicConfiguration (NoteAllocator ci);
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

DISCUSSION

The `NASaveMusicConfiguration` saves the current list of registered devices to a file. This file is read whenever a note allocator connection is opened, restoring the previously configured list of devices. The list is saved in the QuickTime Preferences file.

function result A result code or `-1` if there is a problem opening or creating the QuickTime Preferences file.

NATask

Call the `NATask` function periodically to allow the note allocator to perform tasks in foreground task time.

```
extern pascal ComponentResult NATask (NoteAllocator ci)
```

`ci` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

function result A result code.

DISCUSSION

The `NATask` function calls each registered music component's `MusicTask` function.

Music Component Functions: Synthesizer

The functions in this section obtain specific information about a synthesizer and obtain a best instrument fit for a requested tone from the available instruments within the synthesizer; play a note with a specified pitch, volume, and duration; get and set a particular synthesizer knob; obtain synthesizer knob information; and get and set external MIDI procedure name entry points.

MusicGetDescription

The `MusicGetDescription` function returns a structure describing the synthesizer controlled by the music component device.

```
pascal ComponentResult MusicGetDescription(
                    MusicComponent mc,
                    SynthesizerDescription *sd);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`*sd`	Pointer to synthesizer description structure (page 19-48).
function result	A result code.

DISCUSSION

The `MusicGetDescription` function returns a structure describing the specified music component device. The `SynthesizerDescription` record is filled out by the particular music component.

MusicFindTone

The `MusicFindTone` function returns an instrument number based on a tone description.

```
pascal ComponentResult MusicFindTone(
                    MusicComponent mc,
                    ToneDescription *td,
                    long *instrumentNumber,
                    long *fit);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`*td` Pointer to a tone description structure (page 19-50).

`*instrumentNumber` On exit, contains the number of the best-matching instrument. Only General MIDI numbers are guaranteed to be the same for later instantiations of the component.

`*fit` On exit, indicates how well an instrument matches the tone description. For valid values, see "Tone Fit Flags" (page 19-36).

function result A result code.

DISCUSSION

The `MusicFindTone` function returns the best-matching instrument number for this device. How close a match was attained is returned in "fit".

The music component should search in the following order:

1. If the synthesizer is a General MIDI device, use the `gmNumber`.
2. If `synthesizerType` matches, first try to match `instrumentName`, else try `instrumentNumber`. Failing that, try the `gmNumber`.
3. If `synthesizerType` doesn't match, try the `instrumentName`, then the instrument number.

If none of these rules apply, or the fields are "blank" (zero for the type or numeric fields, or zero-length for the strings) then the call returns instrument 1 and a fit value of zero. The `synthesizerName` field may be ignored by the

component; it is used by the note allocator when deciding which music device to use.

MusicPlayNote

The `MusicPlayNote` function plays a note on a specified part at a specified pitch and velocity.

```
pascal ComponentResult MusicPlayNote(
                    MusicComponent mc,
                    long part,
                    long pitch,
                    long velocity);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	The part to play the note on.
`pitch`	The pitch at which to play the note. Values are 0–127 for MIDI pitch or greater than 65535 for microtonal values.
`velocity`	How hard to strike the key. Values are 0–127 where 0 is silence.
function result	A result code.

DISCUSSION

The `MusicPlayNote` function is used to play notes by their pitch. If the pitch is specified by a number from 0 to 127, it is a MIDI pitch, where 60 is middle-C. If the pitch is a positive number above 65535, the value is a fixed-point pitch value. Thus, microtonal values may be specified.

Velocity refers to how hard the key is struck (if performed on a keyboard-instrument), typically this translates directly to volume, but on many synthesizers this also subtly alters the timbre of the tone.

The current note continues to play until a `MusicPlayNote` with the same pitch and velocity of 0 turns the note off.

MusicGetKnob

The `MusicGetKnob` function returns the value of the specified global synthesizer knob. A global knob controls an aspect of the entire synthesizer. It is not specific to a part within the synthesizer.

```
pascal ComponentResult MusicGetKnob(
                    MusicComponent mc,
                    long knobNumber);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`knobNumber` Knob index or ID.

function result A result code.

MusicSetKnob

The `MusicSetKnob` function modifies the value of the specified global synthesizer knob. A global knob controls an aspect of the entire synthesizer. It is not limited to a part within the synthesizer.

```
pascal ComponentResult MusicSetKnob(
                    MusicComponent mc,
                    long knobNumber,
                    long knobValue);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`knobNumber` Knob index or ID.

`knobValue` Value for specified knob.

function result A result code.

MusicGetKnobDescription

The `MusicGetKnobDescription` function returns a pointer to an initialized knob description structure describing a global synthesizer knob. A global knob controls an aspect of the entire synthesizer; it is not limited to a part within the synthesizer.

```
pascal ComponentResult MusicGetKnobDescription(
                    MusicComponent mc,
                    long knobNumber,
                    KnobDescription *mkd);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`knobNumber` Knob index or ID.

`*mkd` Pointer to a knob description structure (page 19-51).

function result A result code.

DISCUSSION

The initialized `KnobDescription` structure provides the application default values associated with the particular knob. You can use the information returned by a call to the `MusicGetKnobDescription` function to reset a knob to some known, usable value.

MusicGetInstrumentKnobDescription

The `MusicGetInstrumentKnobDescription` function gets the description of an instrument knob.

```
extern pascal ComponentResult MusicGetInstrumentKnobDescription(
                    MusicComponent mc,
                    long knobIndex,
                    KnobDescription *mkd)
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

knobIndex	A knob index or knob ID.
mkd	On exit, a knob description record (page 19-51).
function result	A result code.

DISCUSSION

The MusicGetInstrumentKnobDescription function takes a music component instance identifier in the mc parameter and a knob index or knob ID in the knobIndex parameter. It returns a knob description record in the *mkd parameter.

MusicGetDrumKnobDescription

The MusicGetDrumKnobDescription function returns a description of a drum kit knob.

```
extern pascal ComponentResult MusicGetDrumKnobDescription(
                    MusicComponent mc,
                    long knobIndex,
                    KnobDescription *mkd)
```

mc	Music component instance identifier returned by NAGetRegisteredMusicDevice.
knobIndex	A knob index or knob ID.
*mkd	A pointer to a knob description record (page 19-51).
function result	A result code.

DISCUSSION

The MusicGetDrumKnobDescription function takes a music component in the mc parameter and a knob index or knob ID in the knobIndex parameter. It returns a knob description record in the *mkd parameter.

MusicGetKnobSettingStrings

The `MusicGetKnobSettingStrings` function returns a list of knob setting names known by the specified music component.

```
extern pascal ComponentResult MusicGetKnobSettingStrings(
                    MusicComponent mc,
                    long knobIndex,
                    long isGlobal,
                    Handle *settingsNames,
                    Handle *settingsCategoryLasts,
                    Handle *settingsCategoryNames)
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`knobIndex` The knob index or knob ID.

`isGlobal` If a knob index is used, indicates whether the specified knob is a global knob.

`settingsNames`
The requested list of knob setting strings formatted as a short followed by packed strings.

`settingsCategoryLasts`
A handle containing a group of short integers, the first of which contains the number of shorts to follow.

`settingsCategoryNames`
Knob setting category names formatted as a short followed by a list of names.

function result A result code.

Note
All handles must be disposed of by the caller.

MusicSetMIDIProc

The `MusicSetMIDIProc` function tells the music component what procedure to call when it needs to send MIDI data. This call is implemented only by a music component for a MIDI synthesizer.

```
pascal ComponentResult MusicSetMIDIProc(
                    MusicComponent mc,
                    MusicMIDISendProcPtr MIDISendProc,
                    long refCon);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`MIDISendProc` A pointer to the procedure to use when sending MIDI data.

`refcon` Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.

function result A result code.

MusicGetMIDIProc

The `MusicGetMIDIProc` function returns a pointer to the procedure a music component is using to process external MIDI notes.

```
pascal ComponentResult MusicGetMIDIProc(
                    MusicComponent mc,
                    MusicMIDISendProcPtr *MIDISendProc,
                    long *refCon);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`*MIDISendProc` Pointer to a MIDI serial port call.

`*refcon` Contains a reference constant. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.

function result A result code.

DISCUSSION

The `MusicGetMIDIProc` function returns, in the `*MIDISendProc` parameter, a pointer to the function that processes external MIDI notes. This function was set by a previous call to the `MusicSetMIDIProc` function. If no function has been set with the `MusicSetMIDIProc` function, `MusicGetMIDIProc` returns zero in the `*MIDISendProc` parameter.

MusicGetMIDIPorts

The `MusicGetMIDIPorts` function returns the number of input and output ports a MIDI device has.

```
extern pascal ComponentResult MusicGetMIDIPorts(
                    MusicComponent mc,
                    long *inputPortCount,
                    long *outputPortCount)
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`inputPortCount`
On exit, the number of input MIDI ports available to the music component.

`outputPortCount`
On exit, the number of output MIDI ports available to the music component.

function result A result code.

DISCUSSION

The function takes a music component identifier in the `mc` parameter and returns, in the `inputPortCount` and `outputPortCount` parameters, the number of MIDI input and output ports available to the music component.

This call is implemented only for a hardware synthesizer, such as a NuBus or PCI card device.

MusicSendMIDI

Use the `MusicSendMIDI` function to send a MIDI packet to a specified port.

```
extern pascal ComponentResult MusicSendMIDI(
                    MusicComponent mc,
                    long portIndex,
                    MusicMIDIPacket *mp)
```

`mc`	Music component instance returned by `NAGetRegisteredMusicDevice`.
`portIndex`	The index of the port to send the MIDI packet to. The index value is 1 through the port count returned by the `MusicGetMIDIPorts` function.
`mp`	The music MIDI packet to be sent.
function result	A result code.

DISCUSSION

The `MusicSendMIDI` function takes a music component in the `mc` parameter and a port index in the `portIndex` parameter. It sends the MIDI music packet specified by the `*mp` parameter to the specified port.

This call is implemented only for a hardware synthesizer, such as a NuBus or PCI card device.

MusicGetDeviceConnection

You can use the `MusicGetDeviceConnection` function to find out how many hardware synthesizers are available to a music component and to get the IDs for those devices.

```
extern pascal ComponentResult MusicGetDeviceConnection(
                    MusicComponent mc,
                    long index,
                    long *id1,
                    long *id2)
```

`mc`	Music component returned by `NAGetRegisteredMusicDevice`.
`index`	Index of the device for which you want to find out the IDs. Set to 0 if you are calling to get the number of hardware devices.
`id1`	On exit, a hardware synthesizer ID.
`id2`	On exit, another hardware synthesizer ID.
function result	A result code.

DISCUSSION

To get the number of hardware synthesizers available to the music component specified in the `mc` parameter and an index you can use to request ID numbers for a specific device, call the `MusicGetDeviceConnection` function with a value of 0 for the `index` parameter. You can then pass an index value in the `index` parameter, and the function returns hardware synthesizer IDs in the `*id1` and `*id2` parameters.

This call is implemented only for a hardware synthesizer, such as a NuBus or PCI card device.

MusicUseDeviceConnection

The `MusicUseDeviceConnection` function tells a music component which hardware synthesizer to talk to.

```
extern pascal ComponentResult MusicUseDeviceConnection(
                    MusicComponent mc,
                    long id1,
                    long id2)
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`id1` The ID of the device returned in the `*id1` parameter of the `MusicGetDeviceConnection` function.

`id2` The ID of the device returned in the `*id2` parameter of the `MusicGetDeviceConnection` function.

function result A result code.

DISCUSSION

This call is implemented only for a hardware synthesizer, such as a NuBus or PCI card device.

Music Component Functions: Instruments and Parts

The functions described in this section initialize a part with an instrument, store instruments, list available instruments, manipulate parts, and get information about parts.

MusicGetPartInstrumentNumber

The `MusicGetPartInstrumentNumber` function returns the instrument number currently assigned to that part.

```
pascal ComponentResult MusicGetPartInstrumentNumber(
                    MusicComponent mc,
                    long part);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`part` Part number containing instrument.

function result A positive return value is the instrument number; a negative value is a result code.

MusicSetPartInstrumentNumber

The `MusicSetPartInstrumentNumber` function initializes a part with a particular instrument.

```
pascal ComponentResult MusicSetPartInstrumentNumber(
                    MusicComponent mc,
                    long part,
                    long instrumentNumber);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`part` Part to be initialized.

`instrumentNumber`
Number of instrument to initialize part with.

function result A result code.

DISCUSSION

You can use the `MusicFindTone` function (page 19-104) to find out an instrument number.

MusicGetPartAtomicInstrument

The `MusicGetPartAtomicInstrument` function returns the atomic instrument currently in a part.

```
extern pascal ComponentResult MusicGetPartAtomicInstrument(
                    MusicComponent mc,
                    long part,
                    AtomicInstrument *ai,
                    long flags)
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	The part with the atomic instrument.
`ai`	On exit, an atomic instrument.
`flags`	Specify what pieces of information about an atomic instrument the caller is interested in. See "Atomic Instrument Information Flags" on page 40.
function result	A result code.

MusicSetPartAtomicInstrument

The `MusicSetPartAtomicInstrument` function initializes a part with an atomic instrument.

```
extern pascal ComponentResult MusicSetPartAtomicInstrument(
                    MusicComponent mc,
                    long part,
                    AtomicInstrumentPtr aiP,
                    long flags)
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	The part to initialize with the atomic instrument to.
`aiP`	The atomic instrument.

`flags` These flags specify details of initializing a part with an atomic instrument. See "Setting Atomic Instruments" on page 41.

function result A result code.

MusicStorePartInstrument

The `MusicStorePartInstrument` function puts whatever instrument is on the specified part into the synthesizer's instrument store. This enables you to store modified instruments.

```
pascal ComponentResult MusicStorePartInstrument(
                    MusicComponent mc,
                    long part,
                    long instrumentNumber);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`part` Part containing the instrument to be stored.

`instrumentNumber`
Instrument number at which to store the part.

function result A result code.

DISCUSSION

The value of the `InstrumentNumber` parameter must be between 1 and the synthesizer's modifiable instrument count, as defined by the `modifiableInstrumentCount` field of the synthesizer's description record.

MusicGetInstrumentAboutInfo

The `MusicGetInstrumentAboutInfo` function gets the information about an instrument that appears in its About box.

```
pascal ComponentResult MusicGetInstrumentAboutInfo(
                    MusicComponent mc,long part,
                    InstrumentAboutInfo *iai);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`part` Number of the part containing the instrument for which you want information.

`*iai` On exit, a pointer to an instrument About information structure (page 19-52) for the instrument currently on the specified synthesizer part.

MusicGetInstrumentInfo

The `MusicGetInstrumentInfo` function gets a list of instruments supported by a synthesizer. It also gets the names of the instruments.

```
extern pascal ComponentResult MusicGetInstrumentInfo(
                    MusicComponent mc,
                    long getInstrumentNamesFlags,
                    InstrumentInfoListHandle *infoListH)
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`getInstrumentNamesFlags`
Use these flags to specify whether you want a list of fixed instruments, modifiable instruments, or all instruments. See "Instrument Info Flags" (page 19-41).

`infoListH` On exit, the list of instruments (page 19-53).

function result A result code.

Note
This handle must be disposed of by the caller.

DISCUSSION

The functions takes a music component in the `mc` parameter and instructions regarding which types of instruments to get information for in the `flags` parameter. It returns a handle to an instrument information list in the `*infoListH` parameter.

MusicGetPart

The `MusicGetPart` function returns the MIDI channel and maximum polyphony for a particular part in the `*MIDIChannel` and `*polyphony` parameters.

```
pascal ComponentResult MusicGetPart(
                    MusicComponent mc,
                    long part,
                    long *MIDIChannel,
                    long *polyphony)
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	The music component part requested.
`*MIDIChannel`	Pointer to long for `MIDIChannel` result.
`*polyphony`	Pointer to long for polyphony result.
function result	A result code.

DISCUSSION

For non-MIDI devices, the MIDI channel pointed to by the `MIDIChannel` parameter is 0.

MusicSetPart

The `MusicSetPart` function sets the MIDI channel and maximum polyphony for the specified part to the values in the `MIDIChannel` and `polyphony` parameters.

```
pascal ComponentResult MusicSetPart(
                    MusicComponent mc,
                    long part,
                    long MIDIChannel,
                    long polyphony)
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	Part whose MIDI channel and polyphony are to be set.
`MIDIChannel`	The MIDI channel to set the part to.
`polyphony`	The maximum voices or polyphony for the part.
function result	A result code.

DISCUSSION

For non-MIDI devices, set the MIDI channel pointed to by the `MIDIChannel` parameter to 0.

MusicGetPartName

The `MusicGetPartName` function returns the string name of a part.

```
pascal ComponentResult MusicGetPartName(
                    MusicComponent mc,
                    long part,
                    Str31 name);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	Part to get name of.

`name`	On exit, the string containing the part name.
function result	A result code.

DISCUSSION

The name string is used by selection dialogs or configuration information.

MusicSetPartName

You can use the `MusicSetPartName` function to change the name of an instrument in a specified part. For example, you might want to change the name of a modified instrument before saving it.

```
pascal ComponentResult MusicSetPartName(
                    MusicComponent mc,
                    long part,
                    Str31 name);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	Part to apply name to.
`name`	Name to apply to part.
function result	A result code.

DISCUSSION

The instrument name string is used by selection dialogs or in configuration information.

MusicGetPartKnob

The `MusicGetPartKnob` function gets the current value of a knob for a part.

```
pascal ComponentResult MusicGetPartKnob(
                    MusicComponent mc,
                    long part,
                    long knobNumber);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	The part number.
`knobNumber`	The knob index or ID.
function result	Positive or negative integers are knob values. Result codes are returned as `0x8000xxxx`, where `xxxx` is the result code.

MusicSetPartKnob

The `MusicSetPartKnob` function sets a knob for a specified part.

```
pascal ComponentResult MusicSetPartKnob(
                    MusicComponent mc,
                    long part,
                    long knobNumber,
                    long knobValue);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	The part number.
`knobNumber`	The index or ID of the knob to be set.
`knobValue`	The value to set the knob to.
function result	A result code.

MusicResetPart

The `MusicResetPart` function silences all sounds on the specified part, and resets all controllers on that part to their default values. The default value is zero for all controllers except volume. Volume is set to its maximum 32767 or, in hexadecimal, `7F.FF`.

```
pascal ComponentResult MusicResetPart(
                    MusicComponent mc,
                    long Part);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`part` The number of the part.

function result A result code.

MusicGetPartController

The `MusicGetPartController` function returns the value of the specified controller on the specified part.

```
pascal ComponentResult MusicGetPartController(
                    MusicComponent mc,
                    long part,
                    long controllerNumber);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`part` Part whose controller value you want to get.

`controllerNumber`
Controller number.

function result A result code.

MusicSetPartController

The `MusicSetPartController` function initializes the value of the specified controller on the specified part.

```
pascal ComponentResult MusicSetPartController(
                    MusicComponent mc,
                    long part,
                    long controllerNumber,
                    long controllerValue);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`part` Part to apply controller to.

`controllerNumber`
Controller number. For valid values see "Controller Numbers" (page 19-33).

`controllerValue`
Value for controller.

function result A result code.

MusicSetPartSoundLocalization

The `MusicSetPartSoundLocalization` function passes sound localization data to a specified synthesizer part.

```
extern pascal ComponentResult MusicSetPartSoundLocalization(
                    MusicComponent mc,
                    long part,
                    Handle data)
```

`mc` Music component instance identifier.

`part` The part to pass the data to.

`data` The sound localization data.

function result A result code.

Music Component Functions: Miscellaneous

Use the functions described in this section to get and modify the master tuning of the synthesizer, to play off-line, and to allow the music component to perform tasks it must perform at foreground task time.

MusicGetMasterTune

The `MusicGetMasterTune` function returns a fixed-point value in semitones, which is the synthesizer's master tuning.

```
pascal ComponentResult MusicGetMasterTune (MusicComponent mc);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

function result The function returns a positive value representing the synthesizer's master tuning or a negative result code.

MusicSetMasterTune

The `MusicSetMasterTune` function alters the synthesizer's master tuning.

```
pascal ComponentResult MusicSetMasterTune(
                    MusicComponent mc,
                    Fixed masterTune);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`masterTune` The amount by which to transpose the entire synthesizer in pitch. The value is a fixed 16.16 number that allows shifts by fractional values.

function result A result code.

MusicStartOffline

The `MusicStartOffline` function informs the QuickTime music synthesizer that the music will not be played through the speakers. Instead, audio data will be sent to a function that will create a sound file to be played back later.

```
extern pascal ComponentResult MusicStartOffline(
                    MusicComponent mc,
                    unsigned long *numChannels,
                    UnsignedFixed *sampleRate,
                    unsigned short *sampleSize,
                    MusicOfflineDataUPP dataProc,
                    long dataProcRefCon)
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`numChannels` Number of channels in the music sample. 1 indicates monaural; 2 indicates stereo.

`sampleRate` The number of samples per second.

`sampleSize` The size of the music sample: 8-bit or 16-bit.

`dataProc` A function to handle the audio data.

`dataProcRefCon` A reference constant to pass to the `dataProc` function.

function result A result code.

DISCUSSION

You pass the `MusicStartOffline` function the requested values for the `numChannels`, `sampleRate`, and `sampleSize` parameters. When the function returns, those parameters contain the actual values used.

MusicSetOfflineTimeTo

The `MusicSetOfflineTimeTo` function advances the synthesizer clock when the synthesizer is not running in real time (due to a call to `MusicStartOffline`).

```
extern pascal ComponentResult MusicSetOfflineTimeTo(
                    MusicComponent mc,
                    long newTimeStamp)
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`newTimeStamp`	The number of samples to synthesize.
function result	A result code.

DISCUSSION

Setting the time generates audio output from the synthesizer.

MusicTask

Call the `MusicTask` function periodically to allow a music component to perform tasks it must perform at foreground task time.

```
extern pascal ComponentResult MusicTask (MusicComponent mc)
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
function result	A result code.

DISCUSSION

In the case of the QuickTime Music Synthesizer, instruments cannot be loaded from disk at interrupt time, so if the `NASetInstrumentNumberInterruptSafe` function is called, the instrument is loaded during the next `MusicTask` call.

Instrument Component Functions

This section describes functions that are implemented by instrument components.

InstrumentGetInfo

The `InstrumentGetInfo` function returns information about all the atomic instruments supported by an instrument component.

```
extern pascal ComponentResult InstrumentGetInfo(
                    ComponentInstance ci,
                    InstCompInfoHandle *instInfo)
```

`ci` The instrument component instance. You obtain the identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`instInfo` On exit, an instrument information list (page 19-56).

function result A result code.

InstrumentGetInst

The `InstrumentGetInst` function returns an atomic instrument.

```
extern pascal ComponentResult InstrumentGetInst(
                    ComponentInstance ci,
                    long instID,
                    AtomicInstrument *atomicInst,
                    long flags)
```

`ci` The instrument component instance. You obtain the identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`instID`	The instrument component instrument ID from the information list structure returned by the `InstrumentGetInfo` function.
`atomicInst`	On exit, the atomic instrument.
`flags`	Specify what pieces of information about an atomic instrument the caller is interested in. See "Atomic Instrument Information Flags" on page 40.
function result	A result code.

InstrumentInitialize

Used by developers of instrument components, this is a call the instrument component makes to the base class instrument component to tell it how to interpret the instrument component resources.

```
extern pascal ComponentResult InstrumentInitialize(
                    ComponentInstance ci,
                    long initFormat,
                    void *initParams)
```

`ci`	An instrument component instance. You obtain the identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.
`initFormat`	Set to zero.
`initParams`	Set to NULL.
function result	A result code.

InstrumentOpenComponentResFile

The `InstrumentOpenComponentResFile` function opens the resource file containing the instruments in the instrument component and makes it the current resource file.

```
extern pascal ComponentResult InstrumentOpenComponentResFile(
                    ComponentInstance ci,
                    short *resFile)
```

`ci` The instrument component instance. You obtain the identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`resFile` On exit, a resource reference.

function result A result code.

InstrumentCloseComponentResFile

The `InstrumentCloseComponentResFile` function closes a resource file.

```
extern pascal ComponentResult InstrumentCloseComponentResFile(
                    ComponentInstance ci,
                    short resFile)
```

`ci` The instrument component instance. You obtain the identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`resFile` A reference to the resource file that was returned previously by the `InstrumentOpenComponentResFile` function.

function result A result code.

InstrumentGetComponentRefCon

The `InstrumentGetComponentRefCon` function gets the reference constant for an instrument component.

```
extern pascal ComponentResult InstrumentGetComponentRefCon(
                    ComponentInstance ci,
                    void **refCon)
```

`ci` The instrument component instance. You obtain the identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`refCon` A reference constant.

function result A result code.

InstrumentSetComponentRefCon

Use the `InstrumentSetComponentRefCon` function to override the Component Manager `SetComponentRefCon` function and set the instrument component's reference constant to a specified value.

```
extern pascal ComponentResult InstrumentSetComponentRefCon(
                    ComponentInstance ci,
                    void *refCon)
```

`ci` The instrument component instance. You obtain the identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for details.

`refCon` A reference constant.

function result A result code.

Result Codes

This section lists all the result codes returned by QuickTime Music Architecture functions.

NOTIMPLEMENTEDMUSICOSERR	-2071	Call to a routine that is not supported by a particular music component.
CANTSENDTOSYNTHESIZEROSERR	-2072	Attempt to use a synthesizer before it has been initialized, given a MIDI port to use, or told which slot card to use. For example, the MusicSetMIDIProc function has not been called.
ILLEGALVOICEALLOCATIONOSERR	-2074	Attempt to allocate more voices than a synthesizer supports.
ILLEGALPARTOSERR	-2075	Usually indicates use of a part number parameter outside the range 1…partcount
ILLEGALCHANNELOSERR	-2076	Attempt to use a MIDI channel outside the range 1…16
ILLEGALKNOBOSERR	-2077	Attempt to use a knob index or knob ID that is not valid
ILLEGALKNOBVALUEOSERR	-2078	Attempt to set a knob outside its allowable range, as specified in its knob description record.
ILLEGALINSTRUMENTOSERR	-2079	Attempt to use an instrument or sound that is not available or there is some other problem with the instrument, such as a bad instrument number.
ILLEGALCONTROLLEROSERR	-2080	Attempt to get or set a controller that is outside the allowable controller number range or is not recognized by this particular music component.
MIDIMANAGERABSENTOSERR	-2081	Attempt to use MIDI Manager for a synthesizer when the MIDI Manager is not installed.
SYNTHESIZERNOTRESPONDINGOSERR	-2082	Various hardware problems with a synthesizer.
SYNTHESIZEROSERR	-2083	Software problem with a synthesizer.

ILLEGALNOTECHANNELOSERR	-2084	Attempt to use a note channel that is not initialized or is otherwise errant.
NOTECHANNELNOTALLOCATEDOSERR	-2085	It was not possible to allocate a note channel.
TUNEPLAYERFULLOSERR	-2086	Attempt to queue up more tune segments (with `TuneQueue`) than allowed.
TUNEPARSEOSERR	-2087	`TuneSetHeader` or `TuneQueue` encountered an absurd bit of tune sequence data.

General MIDI Reference

General MIDI Instrument Numbers

Table A-1 General MIDI Instrument Numbers

1	Acoustic Grand Piano	33	Wood Bass
2	Bright Acoustic Piano	34	Electric Bass Fingered
3	Electric Grand Piano	35	Electric Bass Picked
4	Honky-tonk Piano	36	Fretless Bass
5	Rhodes Piano	37	Slap Bass 1
6	Chorused Piano	38	Slap Bass 2
7	Harpsichord	39	Synth Bass 1
8	Clavinet	40	Synth Bass 2
9	Celesta	41	Violin
10	Glockenspiel	42	Viola
11	Music Box	43	Cello
12	Vibraphone	44	Contrabass
13	Marimba	45	Tremolo Strings
14	Xylophone	46	Pizzicato Strings
15	Tubular bells	47	Orchestral Harp
16	Dulcimer	48	Timpani
17	Draw Organ	49	Acoustic String Ensemble 1
18	Percussive Organ	50	Acoustic String Ensemble 2

Table A-1 General MIDI Instrument Numbers (continued)

19	Rock Organ	51	Synth Strings 1
20	Church Organ	52	Synth Strings 2
21	Reed Organ	53	Aah Choir
22	Accordion	54	Ooh Choir
23	Harmonica	55	Synvox
24	Tango Accordion	56	Orchestra Hit
25	Acoustic Nylon Guitar	57	Trumpet
26	Acoustic Steel Guitar	58	Trombone
27	Electric Jazz Guitar	59	Tuba
28	Electric clean Guitar	60	Muted Trumpet
29	Electric Guitar muted	61	French Horn
30	Overdriven Guitar	62	Brass Section
31	Distortion Guitar	63	Synth Brass 1
32	Guitar Harmonics	64	Synth Brass 2
65	Soprano Sax	97	Ice Rain
66	Alto Sax	98	Soundtracks
67	Tenor Sax	99	Crystal
68	Baritone Sax	100	Atmosphere
69	Oboe	101	Bright
70	English Horn	102	Goblin
71	Bassoon	103	Echoes
72	Clarinet	104	Space
73	Piccolo	105	Sitar
74	Flute	106	Banjo
75	Recorder	107	Shamisen

Table A-1 General MIDI Instrument Numbers (continued)

76	Pan Flute	108	Koto
77	Bottle blow	109	Kalimba
78	Shakuhachi	110	Bagpipe
79	Whistle	111	Fiddle
80	Ocarina	112	Shanai
81	Square Lead	113	Tinkle bell
82	Saw Lead	114	Agogo
83	Calliope	115	Steel Drums
84	Chiffer	116	Woodblock
85	Synth Lead 5	117	Taiko Drum
86	Synth Lead 6	118	Melodic Tom
87	Synth Lead 7	119	Synth Tom
88	Synth Lead 8	120	Reverse Cymbal
89	Synth Pad 1	121	Guitar Fret Noise
90	Synth Pad 2	122	Breath Noise
91	Synth Pad 3	123	Seashore
92	Synth Pad 4	124	Bird Tweet
93	Synth Pad 5	125	Telephone Ring
94	Synth Pad 6	126	Helicopter
95	Synth Pad 7	127	Applause
96	Synth Pad 8	128	Gunshot

General MIDI Drum Kit Numbers

Table A-2 General MIDI Drum Kit Numbers

35	Acoustic Bass Drum	51	Ride Cymbal 1
36	Bass Drum 1	52	Chinese Cymbal
37	Side Stick	53	Ride Bell
38	Acoustic Snare	54	Tambourine
39	Hand Clap	55	Splash Cymbal
40	Electric Snare	56	Cowbell
41	Lo Floor Tom	57	Crash Cymbal 2
42	Closed Hi Hat	58	Vibraslap
43	Hi Floor Tom	59	Ride Cymbal 2
44	Pedal Hi Hat	60	Hi Bongo
45	Lo Tom Tom	61	Low Bongo
46	Open Hi Hat	62	Mute Hi Conga
47	Low -Mid Tom Tom	63	Open Hi Conga
48	Hi Mid Tom Tom	64	Low Conga
49	Crash Cymbal 1	65	Hi Timbale
50	Hi Tom Tom	66	Lo Timbale

General MIDI Kit Names

Table A-3 General MIDI Kit Names

1	Dry Set
9	Room Set
19	Power Set
25	Electronic Set
33	Jazz Set
41	Brush Set
65-112	User Area
128	Default

QuickTime File Format Changes

This appendix contains changes and additions to the Motion JPEG and YUV file formats as documented in *QuickTime File Format Specification, May 1996*. Information about the QuickTime image file format introduced with QuickTime 2.5 is also included.

Motion JPEG

M-JPEG Format A

The following two fields have been added to format A:

Field descriptions

Start of scan offset

Specifies the offset, in bytes, from the start of the field data to the start of the scan marker. This field should never be set to 0.

Start of data offset

Specifies the offset, in bytes, from the start of the field data to the start of the data stream. Typically this immediately follows the start of scan data.

M-JPEG Format B

The following two fields have been added to format B:

Field descriptions

Start of scan offset

Specifies the offset, in bytes, from the start of the field data to the contents of the start of scan data. This field should never be set to 0.

Start of data offset

Specifies the offset, in bytes, from the start of the field data to the start of the data stream. Typically this immediately follows the start of scan data.

YUV

QuickTime 1.6.1 introduced the Component Video codec, which stores data in YUV 4:2:2 format. The compression algorithm is not lossless, but the image quality is extremely high. The compression ratio is 3:2 (or 1.5:1). It does not support frame differencing. This codec is useful for certain video input solutions, such as those included in the Macintosh Quadra and Power Macintosh AV models. The YUV format is also useful as an intermediate storage format if you are applying multiple effects or transitions to an image.

By default, the Component Video compressor does not appear in the Standard Compression dialog box. However, it will appear if you hold down the Option key when clicking the compressor list to display the complete list.

Uncompressed YUV2

The YUV2 stream is encoded in a series of four-byte packets. Each packet represents two adjacent pixels on the same scan line. The bytes within each packet are ordered as follows:

```
y0 u y1 v
```

`y0` is the luminance value for the left pixel; `y1` the luminance for the right pixel. `u` and `v` are chromatic values that are shared by both pixels. The conversion into RGB space is represented by the following equations:

$r = 1.402 * v + y + .5$

$g = y - .7143 * v - .3437 * u + .5$

$b = 1.77 * u + y + .5$

The r, g, and b values range from 0 to 255.

QuickTime Image File Format

QuickTime image files are intended to provide the most useful container for QuickTime compressed still images. The format uses the same atom-based structure as a QuickTime movie. There are two defined atom types: `'idsc'`, which contains an image description, and `'idat'`, which contains the image data. For a JPEG image, the image description atom contains a QuickTime image description describing the JPEG image's size, resolution, depth, and so on, and the image data atom contains the actual JPEG compressed data. A QuickTime image file can also contain other atoms. For example, it can contain single-fork preview atoms. Because the QuickTime image file is a single fork format, it works well in cross-platform applications. On MacOS systems, QuickTime image files are identified by the file type `'qtif'`. Apple recommends using the filename extension `.QIF` to identify QuickTime image files on other platforms.

Glossary

THIS IS A PROTOTYPE GLOSSARY !!

ambient Of a sound, to appear to be emanating from all directions. Compare **binaural, localized.**

angular attenuation The loss of a sound's volume due to a change in the angle between the sound source orientation and the vector between the source and the listener. Compare **distance attenuation, reverberation attenuation, room reflectivity attenuation.**

angular attenuation cone A cone that determines the direction of maximum sound intensity and the amount of attenuation that occurs as the angle between the orientation vector and the source-to-listener vector increases toward a predefined limit.

attenuation The loss of sound volume caused by some physical action on the sound (such as its traveling over a distance or reverberating off a wall). See also **angular attenuation, distance attenuation, reverberation attenuation, room reflectivity attenuation.**

axis element A continuous control element with or without a meaningful center—for example, a joystick or a gas pedal. See also **button element, directional pad element.**

back buffer The buffer DrawSprocket draws into while another image buffer is being displayed.

bilinear interpolation Bilinear interpolation averages the pixels and determines a pixel value that falls between the two original pixels.

binaural Of a sound, recorded with a localized effect. Compare **ambient, localized.**

blanking window A window with which DrawSprocket completely covers the display, hiding the desktop, menu bar, and other system resources, and providing a uniform background color for the game to draw over.

button element A two-state element. See also **axis element, directional pad element, movement element.**

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W, X, Y, Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro printer. Final pages were created on a Docutek. Line art was created using Adobe™ Illustrator and Adobe Photoshop. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITING MANAGERS
Trish Eastman, Michael Hinkson

LEAD WRITER
Linda Kyrnitszke

WRITERS
Judy Helfand, Mimi Jones, Linda Kyrnitszke, Gary McCue, Lori Stipp, Mark Turner

DEVELOPMENTAL EDITOR
Wendy Krafft

ILLUSTRATOR
Deb Dennis

PRODUCTION EDITORS
Pat Christenson, Gerri Gray, Alan Morgenegg

Special thanks to Sean Allen, Drew Colace, Peter Hoddie, Ian Ritchie, and Charles Wiltgen

Developer Press Art List

QuickTime 2.5 Delta Guide

Writer	*Mark Turner*	Art Director	*name*
Prod. Editor	*Pat Christenson*	Illustrator	*name*
# of Figures	*31*	Draft stage	*DR_seed_alpha_beta_final*
